НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

"На правах рукопису"
УДК_____

«До захисту допущено»
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
"___"_____2018р.

Магістерська дисертація

зі спеціальності 121 Інженерія програмного забезпечення
за спеціалізацією Програмне забезпечення розподілених систем
на тему "Програмні засоби прискорення модульного тестування"

Виконав: студент 6 курсу, групи ТВз.-71мп
Ігушкіна Тетяна Сергіївна _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник к.т.н, доцент Смаковський Д. С. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний
(повна назва)

Кафедра автоматизації проектування енергетичних процесів і систем
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
«___»_____20___ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Ігушкіної Тетяна Сергіївни
(прізвище, ім’я, по батькові)

1. Тема дисертації: Програмні засоби прискорення модульного тестування
науковий керівник дисертації Смаковський Денис Сергійович, к.т.н. доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від « » 2018 р. № _____

2. Термін подання студентом дисертації: 10 грудня 2018 року

3. Об’єкт дослідження: Засоби прискорення модульного тестування

4. Предмет дослідження: Засоби модульного тестування

5. Перелік питань, які потрібно розробити:
5.1. Аналіз сучасних методів модульного тестування.
5.2. Аналіз модулів системи, що придатні для модульного тестування.
5.3 Аналіз шляхів оптимізації модульного тестування.
5.4. Розробка оптимізованого модулю для тестування.
5.5. Тестування розробленого модуля на результативність.
5.6. Розробка засобів прискорення модульного тестування.
5.7. Розробка стартап-проекту.

6. Орієнтовний перелік ілюстративного матеріалу:

6.1. Презентація PowerPoint відповідно до теми дисертації.

7. Дата видачі завдання «__11_ »_____вересня_____2018р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
	Затвердження теми роботи	17.05.2018	
	Вивчення та аналіз задачі. Проведення дослідження по вибраній темі	01.06.2018-03.09.2018	
	Розробка архітектури та загальної структури системи	03.09.2018-28.09.2018	
	Програмна реалізація системи	01.10.2018-26.10.2018	
	Захист програмного продукту	22.10.2018	
	Оформлення пояснювальної записки	02.09.2018-10.12.2018	
	Передзахист	26.11.2018-30.11.2018	
	Захист	19.12.2018	

Студент _____ Ігушкіна Т.С._____
(підпис) (прізвище та ініціали)

Науковий керівник _____ Смаковський Д.С._____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Структура та обсяг дипломної роботи

Магістерська дисертація складається зі вступу, шести розділів, висновку, переліку посилань з 42 найменувань, 4 додатки, і містить 11 рисунків, 20 таблиць. Повний обсяг магістерської дисертації складає 96 сторінок, з яких перелік посилань займає 2 сторінки, додатки – 6 сторінок.

Актуальність теми. Якість розробленого продукту — характеристика, максимального рівня якої прагнуть досягнути при розробці програмного забезпечення. З цією метою, для виявлення дефектів якомога раніше, застосовують модульне тестування. Модульні тести запускаються щоразу, коли розробник вносить зміни в проект — і щоразу займають час. Тому постає задача зменшити час на виконання модульних тестів.

Мета дослідження полягає в розробці засобів, що прискорять виконання модульних тестів за рахунок їх паралельного запуску.

Для досягнення поставленої задачі були сформульовані наступні **завдання:**

- проаналізувати проблеми модульного тестування;
- дослідити існуючі рішення та засоби модульного тестування;
- спроектувати програмні засоби для прискорення модульного тестування;
- розробити рішення, яке дозволить виконувати модульні тести паралельно;

Об'єктом дослідження є програмне забезпечення автоматизованих систем;

Предметом дослідження є програмні засоби прискорення процесів модульного тестування.

Методи дослідження. Розв'язання поставлених задач виконувались, зокрема з використанням наступних методів наукового дослідження:

— формалізація;

— синтез.

Наукова новизна одержаних результатів. Найбільш суттєвими науковими результатами магістерської дисертації є:

— удосконалено програмне рішення щодо процесів модульного тестування, яке полягає у зменшенні часу на виконання модульних тестів завдяки паралельному їх запуску за допомогою розширеної бібліотеки;

— набуло подальшого розвитку використання запропонованого рішення у реальному проекті.

Практичне значення одержаних результатів роботи полягає в розробці програмних засобів, застосування яких прискорює час на виконання модульних тестів, раціоналізує використання ресурсів, завдяки чому заощаджує кошти.

Апробація результатів дисертації

Основні положення роботи доповідались і обговорювались на :

1. V Міжнародній науково-практичній конференції «Сталий розвиток XXI століття: управління, технології, моделі» (м. Київ, 23-24 жовтня 2018 року).

Публікації. Наукові положення дипломної роботи опубліковані у 1 монографії.

Ключові слова. МОДУЛЬНЕ ТЕСТУВАННЯ, БАГАТОПОТОЧНІСТЬ, РЕФЛЕКСІЯ, ФОРК-ДЖОЙН.

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

BDD	Behavior-driven development – Розробка через поведінку
Fork Join	(Форк Джойн) - метод, застосовуваний для збільшення продуктивності виконання великої кількості робочих завдань, полягає в тому, що кожна задача розбивається на безліч дрібніших синхронізованих завдань, які обробляються паралельно на різних серверах.
IDE	Integrated development environment — Інтегроване середовище розробки.
JUnit	Бібілотека для тестування програмного забезпечення для мови Java.
TDD	Test-Driven Development – Розробка через тестування
Unit -тести	Модульні тести – тести, метою яких є ізоляція кожної частини програми та впевненість у тому, що кожна окрема частина є коректною
Рефлексія	(Reflection) - функціонал, який дозволяє отримати опис класу (назви, типи полів, аргументів, методів, а також модифікатори доступу), маючи лише його екземпляр.

ВСТУП

Кожен замовник програмного продукту сплачує кошти, наймає спеціалістів, сподіваючись отримати готовий продукт у найвищій якості, оскільки це може напряму вплинути на його справу. В свою чергу, команда, що займається розробкою, має це розуміти і робити все можливе для задоволення вимог замовника.

В процесах розробки програм існують різні підходи, проте кожен націлений на досягнення максимально якісного кінцевого продукту. В цьому розробникам допомагають різного роду перевірки та аналіз коду. Тест - це процедура, що дозволяє підтвердити або спростувати роботоздатність програмного коду. В ньому містяться перевірки умов, що можуть або виконуватися або не виконуватися. В першому випадку тест вважається пройденим (passed), а в другому - ні (failed). Пройдений тест підтверджує саме ту поведінку кода, яку передбачав розробник. Саме тому тести та їх результати є важливим показником якості готового програмного продукту.

Але важливою є не лише якість, а й можливість досягти її у найраціональніший, найекономічніший, найшвидший спосіб. Тому сьогодні розробники докладають своїх зусиль для створення різних інструментів для спрощення і раціоналізації процесів.

По мірі просування до розподілених, багаторівневих і гетерогенних обчислень технологія J2EE (Java ™ 2 platform, Enterprise Edition) стає найпопулярнішою для розробки заснованих на компонентах, багаторівневих, розподілених корпоративних застосунків. Технологія J2EE інтегрує клієнтські програми і аплети, Web-компоненти (JSP і сервлети) і EJB-компоненти (Enterprise JavaBeans™). Web і EJB-компоненти виконуються на сервері додатків, наприклад, на сервері IBM® WebSphere® Application Server. При використанні технології J2EE для розробки

великих, складних корпоративних додатків стає неминучою ситуація, коли різні компоненти збираються разом для формування єдиного інтегрованого застосунку.

Перед таким компонуванням необхідно і критично важливо виконати належним чином незалежне модульне тестування кожного компонента. Ізольоване модульне тестування кожного модулю значно зменшує кількість помилок і гарантує високу якість програмного забезпечення. Деякі розробники або тестувальники можуть заперечити, що модульне тестування J2EE-компонентів вимагає занадто багато часу і праці, а також схильне до помилок, проте це не так.

Модульний тест являє собою написаний розробником фрагмент коду, який використовує дуже маленьку, спеціалізовану область функціональності тестованого коду. Зазвичай модульний тест виконує деякий конкретний метод в конкретному контексті; таким чином, він потрапляє в більш широку категорію тестування білого ящика. Оскільки необхідність модульного тестування стала очевидною, розробники почали використовувати переваги інтегрованої бібліотеки JUnit для виконання цих тестів. Інтегроване середовище тестування JUnit, спочатку написане Енріке Гамма (Enrich Gamma) і Кентом Беком (Kent Beck), є середовищем для модульного тестування клієнтських Java-додатків. Ця бібліотека пропонує кілька переваг:

- просте інтегроване середовище для написання автоматизованих, самоперевіряючих тестів на Java;
- підтримка тестових тверджень (assertions);
- розробка наборів тестів;
- негайні звіти по тестах.

JUnit надає текстовий командний рядок, а також засновані на AWT і Swing графічні механізми складання звітів з виконанними тестами. Однак для тих розробників, які хочуть проводити модульне тестування в контейнерах сервера додатків і відображати результати в форматі HTML або XML, середа JUnit виявиться обмеженою і неефективною. Тому співтовариство розробників відкритого вихідного коду розширило можливості JUnit, реалізувавши інтегроване середовище

тестування JUnitEE, яка дозволяє запускати модульні JUnit-тести всередині контейнерів сервера додатків.

Також, JUnit не надає можливості запускати модульні тести паралельно. Саме ця проблема лягла в основу завдання цієї роботи - а саме розробити та запропонувати розширення, яке надасть можливість одночасного запуску модульних тестів.

1 АКТУАЛЬНІСТЬ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Сучасні проекти важко уявити без тестування. І це не дивно, адже на різних рівнях можна знайти дефекти та помилки, і, “відловивши” їх, не дозволити недолікам проникнути у поле зору кінцевого користувача. Крім того, процес тестування зазвичай побудований таким чином, що спочатку перевіряються менші частини, потім їх взаємодія, а далі відповідність системи загалом. Це дозволяє від самого початку написання коду знайти дрібніші дефекти, які можна пофіксити відразу та витратити на це менше часу, адже може бути ситуація, що помилку, що дійде до користувача, буде набагато важче виправити, та й вартуватиме це набагато більше. Тест - це оцінка наших знань, доказ концепції або перевірка даних. Тест класу - це аналіз наших знань, щоб переконатися, чи можемо ми перейти на наступний рівень. Для програмного забезпечення це перевірка функціональних та нефункціональних вимог перед тим, як вона відправляється замовнику.

Код модульного тестування означає валідацію чи перевірку правильності коду. Перевірка працездатності - це основний тест, який дозволяє швидко оцінити, чи може результат підрахунку бути дійсним. Це просто перевірити, чи виконана робота є такою, як очікувалося. Поширеною практикою є тестування коду за допомогою операторів друку в основному методі або власне виконання програми. Проте жоден з них не є правильним підходом, оскільки змішування виробничого коду з тестами - не найкраща практика. Тестування логіки у виробничому коді - це погана практика, хоча це й не порушує код під час тесту. Однак це збільшує складність коду і може створити серйозну проблему технічного обслуговування або призвести до відмови системи, якщо щось піде не так. Друковані строки (методи) виконуються у системі та друкують непотрібну інформацію. Вони збільшують час виконання та зменшують читаність коду. Крім того, інформація з журналу інформації може приховати

справжню проблему, наприклад, ви можете не помітити критичного граничного або нагального попередження через надмірний об’єм реєстрації даних.

1.1 Роль модульного тестування

Перевірка якості є невід’ємною частиною будь-якої діяльності. І програмне забезпечення не є виключенням. Для цього використовують тестування на різних рівнях. Найчастіше виділяють наступні рівні тестування (рисунок 1.1): модульне тестування (unit testing), інтеграційне (integration testing), системне (system testing) та приймальне тестування (UAT testing).[1]

Рисунок 1.1. Піраміда рівнів тестування

Кожен рівень є важливим. Рівні тестування полягають у визначенні відсутніх областей та уникненні дублювання та повторення фаз життєвого циклу розробки. У моделях життєвого циклу розробки програмного забезпечення визначаються такі фази, як збирання вимог та аналіз, проектування, кодування або впровадження, тестування та розгортання. На кожному етапі проходить тестування.

Так, модульні тести пишуть розробники, щоб переконатися, що їх код працює нормально та відповідає специфікаціям користувачів. Вони перевіряють свої шматочки коду, такі, наприклад, як класи, функції, інтерфейси та процедури. Ще до готовності програми можна виявити помилки та дефекти в коді. І саме для цього розробники використовують модульне тестування. Запуск модульних тестів проводиться щоразу під час збірки на машині розробника, який час від часу додає певні зміни до проекта.

Для порівняння, інтеграційне тестування виконується, коли об'єднано два модулі, для того, щоб перевірити поведінку та функціональність обох модулів після інтеграції. Іноді можуть виділяти декілька рівнів інтеграційного тестування:
- інтеграційне тестування модулів (компонентів): коли інтегруються модулі, виконується тестування, яке називається тестуванням інтеграції компонентів. Це тестування в основному зроблено для забезпечення того, щоб код не зламався після інтеграції двох модулів;
- тестування системної інтеграції (System integration testing - SIT) - тестування, де тестувальники в основному перевіряють, що в одному середовищі всі пов'язані системи повинні підтримувати цілісність даних і можуть працювати в координації з іншими системами.

А от системне тестування передбачає перевірку всієї системи на наявність помилок та недоліків. Цей тест здійснюється шляхом взаємодії між апаратними та програмними компонентами всієї системи (які раніше були перевірені спочатку модульними, а потім інтеграційними тестами), а потім тестують її в цілому. Цей метод можна віднести до метода тестування "чорного ящику", де програмне забезпечення перевіряється на очікувані користувачем умови роботи, а також на потенційне виключення та граничні умови.

SpiraTest включає підтримку для зберігання, керування та координування системних тестів у всіх модулях та компонентах, що складають систему. SpiraTest (рисунок 1.2) підтримує тестування, засноване на даних, де тестові випадки визначаються з вхідними параметрами, а різні комбінації тестових даних можуть

бути передані до ручних та автоматизованих тестів. Це гарантує, що як очікувані, так і випадки виключення можуть бути перевірені за допомогою однакових тестових фреймворків.

INCLUDEPICTURE "https://lh5.googleusercontent.com/Nw1-rzgdjgZDb_LpDmSExgNUmWuMqfj5lodqJ5CvgfamjMEnSihylNLrS5m6i449jTcv4EhDzFEY8sYZMHFrS7Gck0o2F8xHhcgLIra7UL086IMfXjUJJvUbzRY_hKQMJUmKFqEn" * MERGEFORMATINET

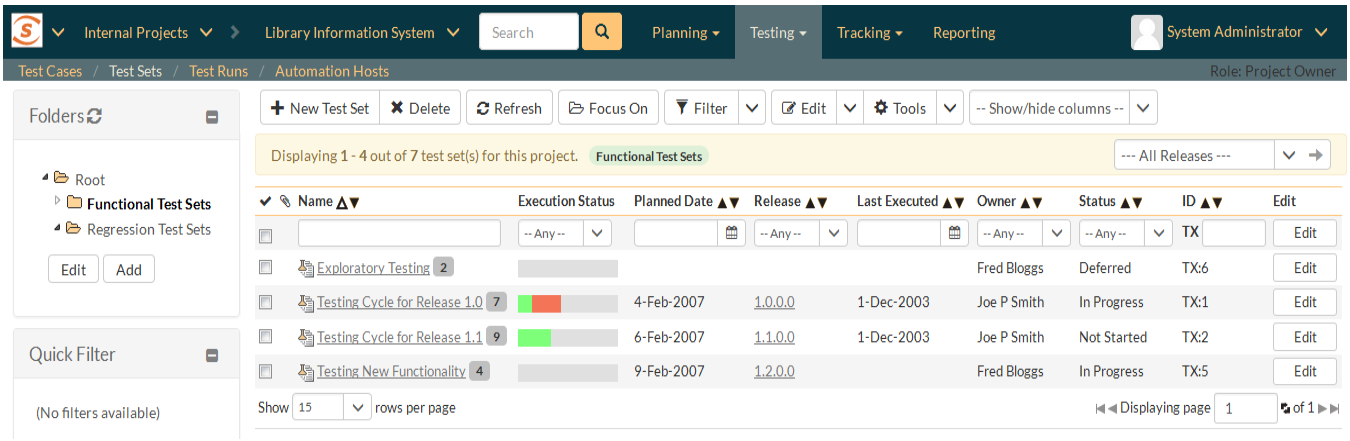


Рисунок 1.2. Знімок екрана про те, як SpiraTest дозволяє вам керувати різними типами тестів.

У тестуванні системи тестувальники загалом перевіряють сумісність програми з системою. Тестування системної інтеграції може виконуватися після тестування системи або паралельно з тестуванням системи.

Приймальні тести здебільшого проводяться для забезпечення відповідності вимогам специфікації. Їх також розділяють на:

- альфа-тестування: проводиться на стороні розробників, це робиться в кінці процесу розробки;
- бета-тестування: проводиться на стороні клієнта, це робиться безпосередньо перед запуском продукту.

Повернемось до модульного тестування та розглянемо, що ж саме потрібно тестувати. Те, що ми тестуємо, розділено на дві загальні категорії: по-перше, це охоплення коду, що має мету виконувати кожен рядок коду у вашій програмі

щонайменше один раз з репрезентативними даними, щоб ви могли бути впевнені, що весь код працює правильно; а по-друге, покриття даних має мету тестування репрезентативних зразків хороших та поганих даних, як вхідних даних, так і даних, створених вашою програмою, з метою забезпечення правильної обробки даних та, зокрема, помилок даних. Звичайно, існує дублювання охоплення коду та покриття даних; іноді для того, щоб отримати певну частину вашої програми для виконання, вам, наприклад, доведеться подавати його неправильні дані.

Перевірка покриття коду (code coverage) полягає в тестуванні кожного твердження у вашій програмі. Щоб це зробити, слід пам'ятати про особливості коду. Тобто програма складається з декількох різних типів коду, кожен з яких потрібно протестувати. По-перше, є прямий код(straight-line code). Прямий код описує один шлях через вашу функцію або метод. Зазвичай це вимагатиме одного тесту на різні типи даних - покриття гілки(branch coverage). За допомогою такого покриття дані тестуються скрізь, де ваша програма може змінити напрямки. Це означає, що тут потрібно подивитися на контрольні структури: а саме кожен оператор if і switch, а також кожне складне умовне вираження - ті, що містять оператори AND і OR.

Для кожного if-твердження, потрібні два тести - один для того, коли умовний вираз істинний, а інший - для помилок. Для кожного перемикання(switch) у методі знадобиться окремий тест для кожної ситуації, що міститься в цьому перемиканні, в тому числі і за замовчуванням (адже всі оператори switch мають стандартний додаток). Логічні оператори AND (&&) і OR (||) додають ускладненість умовних виразів, тому для них потрібні додаткові тестові випадки. В ідеалі буде потрібно чотири тестових випадки для кожного (False-False, False-True, True-False, True-True). Оскільки Java використовує оцінку швидкого доступу для логічних операторів, як і C та C++, то ви можете зменшити число тестових випадків. Для оператора OR вам як і раніше потрібно два випадки, якщо перше висловлювання є помилковим, але ви можете просто використовувати єдиний тестовий випадок, якщо у першій перевірці отримаємо значення true (весь вираз завжди буде вірним). Для оператора AND буде потрібно лише один тест, якщо перший підрозділ оцінює значення false (результат

завжди буде помилковим), але вам потрібні обидва тести, якщо перший підрозділ оцінюється як true. Тоді отримаємо покриття циклу (loop coverage), яке схоже на покриття гілки. Різниця тут полягає в тому, що у режимі for, while, або do-while loops ви маєте найкращу вірогідність введення помилки окремо, і ви повинні точно перевірити це. Крім того, буде потрібний тест на нормальний пробіг через цикл, який також потрібно буде протестувати на пару інших речей. Спочатку з'явиться можливість для попередніх тестів, що ви ніколи не входите в тіло циклу, - умовний вираз петлі не вдається в перший раз. Тоді вам потрібно буде протестувати для нескінченного циклу - умовний вираз ніколи не стає помилковим. Для циклів, які читають файли, зазвичай потрібно перевірити маркер кінцевого файлу (the end-of-file marker - EOF). Це ще одне місце, де помилки можуть відбутися або через передчасний EOF, або тому, що (у випадку використання стандартного вводу) EOF ніколи не вказано. Окрім того, існують також return-значення. У багатьох мовах стандартні бібліотечні функції та операційна система викликають всі повернені значення. Наприклад, в C сімейства функцій fprintf та fscanf повертають кількість символів, надрукованих у вихідний потік, і кількість вхідних елементів, призначених із потоку вводу відповідно. Але майже ніхто не перевіряє значення, що повертаються, хоча це потрібно робити. У Java багато подібних програм повертають значення, які оголошені як пусті(void), а не int, як у C або C ++. Отже, проблема в Java зустрічається рідше, ніж в інших мовах. Однак все ж вона існує. Якщо System.out.print () та System. out.println () в Java обидва оголошені на повернення void, метод System.out.printf () повертає об'єкт PrintStream, який практично повністю ігнорується.[2]

1.2 Поняття методології розробки програмного забезпечення

Виділяють два підходи до розробки через тестування - Test Driven Development і Behavior Driven Development. Концепції обох підходів схожі -

спочатку пишуть тести, а потім йде розробка програмного продукту. Проте TDD більше відноситься до програмування та тестування на рівні технічної реалізації продукта, коли розробники створюють тести. А от в BDD тести описуються звичайною мовою, описуючи поведінку(behaviour). Такий підхід допомагає зменшити кількість документації, особливо, застарілої.

Розробка через тестування зазвичай включає наступні етапи [3]: написання тесту та перевірка (до тих пір, поки перевірка не покаже позитивний результат - тест пройдений), написання коду, запуск всіх тестів та перевірка (якщо тести не проходять, розробник повертається до етапу написання коду), після того, як всі тести пройдуть успішно - можна приступати до покращення коду (рефакторингу) з метою полегшити розуміння коду та майбутні зміни, видалити код, що дублюється. Далі цикл повторюється для наступних нових функціональностей.

Багато команд успішно співпрацюють над створенням та наданням максимально цінного, все більш ефективного та надійного програмного забезпечення. І вони навчаються робити це швидше і ефективніше. Існує ряд методів і прийомів, згрупованих за загальним заголовком "Розробка відповідно до поведінки" (BDD).

BDD допомагає командам зосередити свої зусилля на визначенні, розумінні та створенні корисних функцій, які важливі для бізнесу, і гарантує, що ці функції добре розроблені та добре впроваджені. Практикуючі BDD використовують описи конкретних відомих прикладів поведінки системи, щоб зрозуміти, як функції надають користь бізнесу. BDD заохочує бізнес-аналітиків, розробників програмного забезпечення та тестувальників до більш тісної співпраці, дозволяючи їм висловлювати вимоги більш перевіряючим способом у формі, яку як команда розробників, так і бізнес-партнери можуть легко зрозуміти. Інструменти BDD можуть допомогти перетворити ці вимоги в автоматичні тести, які допоможуть керівництву розробника, перевірити функцію та документувати, що таке додаток. BDD не є самостійно методологією розробки програмного забезпечення. Це не заміна для Scrum, XP, Kanban, RUP або будь-якої іншої методології, яку ви зараз

використовуєте. Як ви побачите, BDD включає, спирається на ідеї та покращує ідеї з багатьох цих методологій. І незалежно від того, яка методологія ви використовуєте, існують способи, за допомогою яких BDD може полегшити процеси.

Тож що BDD нам дає? Візьмемо для прикладу певну компанію, якій потрібен новий модуль для свого бухгалтерського програмного забезпечення. Коли у компанії хочуть додати нову функцію, процес ініціює щось подібне:

1. Компанія розповідає бізнес-аналітику, як вони хочуть, щоб ця функція працювала.
2. Бізнес-аналітик перекладає ці запити у набір вимог для розробників, що описують те, що повинно зробити програма.
3. Розробник перекладає вимоги на тестування коду та одиниці для реалізації нової функції.
4. Тестувальник перекладає вимоги до документа у тести і використовує їх, щоб переконатись, що нова функція відповідає вимогам.
5. Інженери документації потім переводять робоче програмне забезпечення та код назад у звичайну англійську технічну та функціональну документацію.

Є багато можливостей для втрати інформації в перекладі, неправильне розуміння чи просто ігнорування (рисунок 1.3). Швидше за все, сам новий модуль може не робити саме те, що було потрібно, і що документація не відображатиме початкові вимоги, надані спочатку аналітикові. INCLUDEPICTURE "https://lh5.googleusercontent.com/9BQRKkjFgekyxyIMmI7Ptya-xiLWhYahfRO0xt9rEPwRmLVG5iV4OFpiktfEbCRjAB9FoYESLKgWSJoNMUIr5eYZnGTD3whXQDF0SPxGwpBrOX9148e-66zXUmEi8_javCvL2N-U" *

MERGEFORMATINET

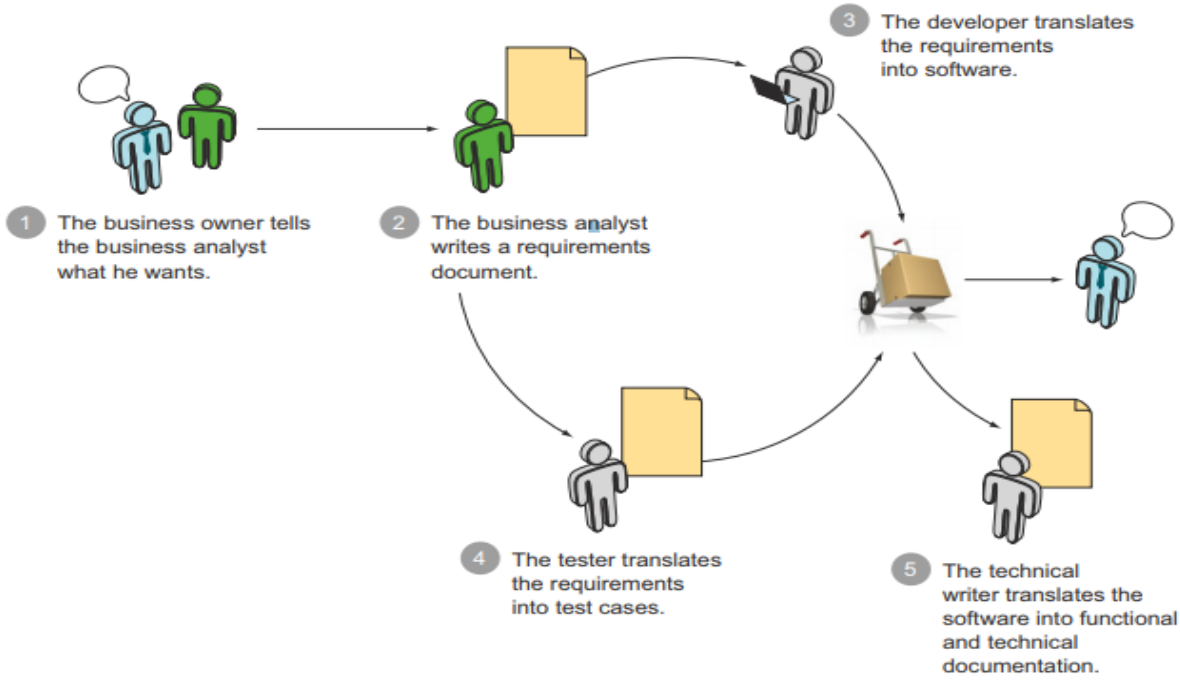


Рисунок 1.3. Схема традиційної моделі розробки.

BDD використовує розмови навколо прикладів, виражений у формі, яку можна легко автоматизувати, щоб зменшити втрачену інформацію та непорозуміння

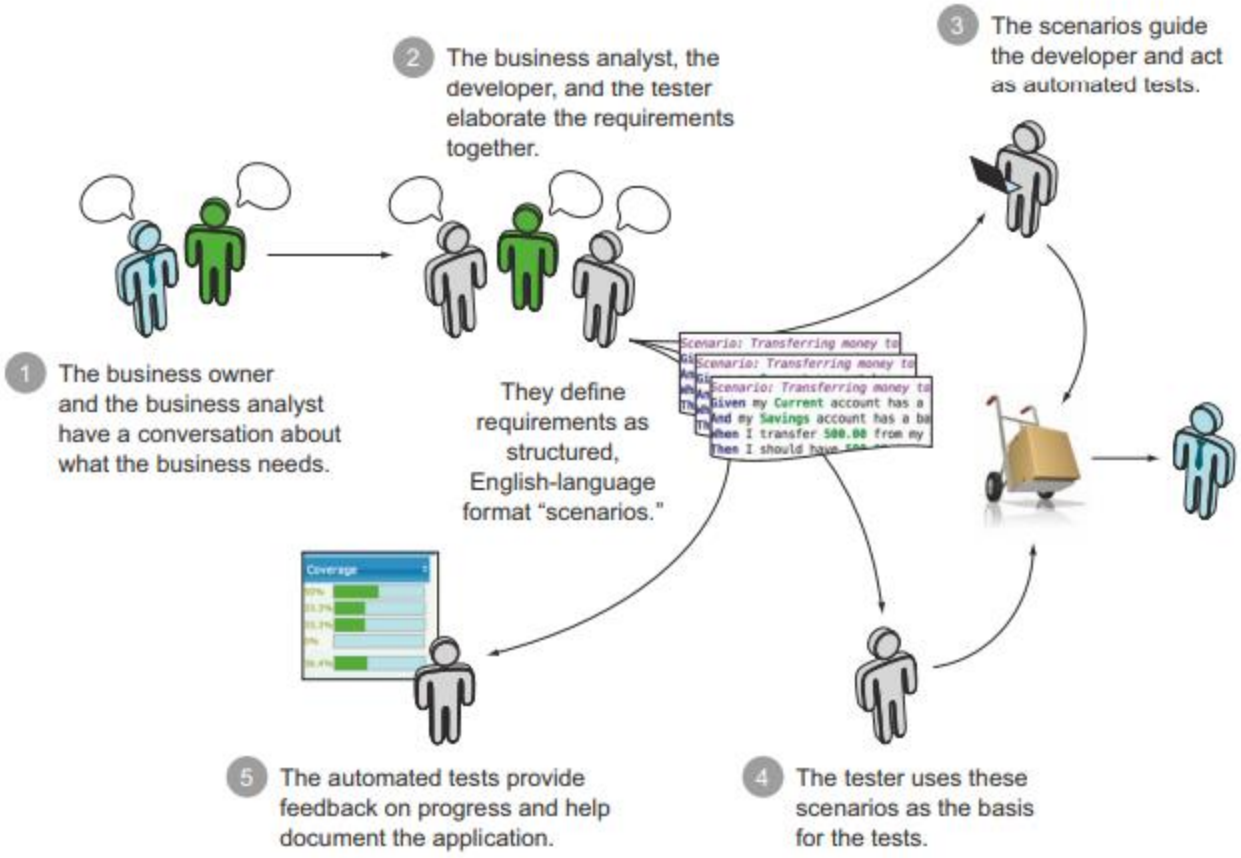


Рисунок 1.4. Модель BDD

Наприклад, поглянемо, як працює інша компанія, яка практикує BDD. У команді бізнес-аналітики, розробники та тестери співпрацюють для розуміння та визначення вимог разом. Вони використовують загальну мову, яка дозволяє легко, неоднозначно вирішувати вимоги кінцевих користувачів до прийнятних, автоматичних тестів. Ці тести визначають, як програмне забезпечення повинно вести себе, і вони керують розробниками у створенні робочого програмного забезпечення, яке зосереджується на функціях, які дійсно важливі для бізнесу. Вони також розмовляють з бізнес-аналітиком про те, що вони хочуть. Щоб зменшити ризик непорозумінь і прихованих припущень, вони говорять конкретними прикладами того, що повинна зробити ця функція.

Перед початком роботи над функцією бізнес-аналітик збирається разом із розробником і тестувальником, що буде працювати над нею, і вони розмовляють про цю функцію. У цій бесіді вони обговорюють та переводять ключові приклади

того, як ця функція повинна працювати в сукупності вимог, написаних в структурованому форматі англійською мовою, який часто називають "Джеркін"(Gherkin). Розробник використовує інструмент BDD, щоб перетворити ці вимоги в набір автоматичних тестів, які працюють із кодом програми та допомагають об'єктивно визначати, коли функція завершена.

Тестування використовує результати цих тестів як відправну точку для ручних та дослідницьких випробувань.

Автоматизовані тести виступають як технічна документація низького рівня та надають найсучасніші приклади роботи системи. Можна переглянути звіти про тести, щоб побачити, які функції були доставлені, і чи вони працюють так, як вона очікувалось. У порівнянні з сценарієм Кріса, команда Сари дуже активно використовує розмови та приклади, щоб зменшити кількість інформації, втраченої при перекладі. Кожен етап після кроку 2 починається зі специфікацій, написаних в Gherkin, які базуються на конкретних прикладах. Таким чином, виникає велика неоднозначність при перекладі початкових вимог клієнта в код, звіти та документацію.

Організації, які охоплюють високоякісні технічні практики, мають іншу історію. Я бачив багато команд, які застосовують такі методи, як тестування, чисте кодування, жива документація та безперервна інтеграція, регулярно звітуючи від низьких до майже нульових коефіцієнтів дефектів, а також код, який набагато простіше адаптуються та поширюються, коли з'являються нові вимоги. і нові функції запитуються. Ці команди також можуть додавати функції більш послідовно, оскільки автоматичні тести гарантують, що існуючі функції не будуть порушені несвідомо. Вони реалізують функції швидше і точніше, ніж інші команди, тому що їм не доведеться боротися за окремий час на виправлення помилок та непередбачуваних побічних ефектів, коли вони вносять зміни. І отримане рішення простіше і дешевше підтримувати.

Ми пишемо тести на все те, що може зламати в основному алгоритм або логіку, наприклад, логіку розрахунку податку на послуги. Але ми не пишемо тести

на очевидні речі, які не можуть вийти з ладу, наприклад, геттери / настанови об'єктів передачі даних або конструкторів або будь-який клас, який просто встановлює значення від одного об'єкта до іншого об'єкта (не виконує жодного перетворення). Але для об'єктів передачі даних, якщо я додаю спеціальну логіку для методів hashCode () або equals (), я обов'язково буду писати тести для перевірки логіки. Тестовий розвиток (TDD) - це новий спосіб програмування. Тут правило дуже просте - треба зробити це наступним чином по кроках:

1. Написати тест, щоб додати нову можливість (автоматизувати тести).
2. Написати код тільки для задоволення умов тестів.
3. Повторно запустити тести - якщо будь-який тест не працює, повернути зміни
4. Зробити рефакторінг та переконатися, що всі тести є “зеленими”.
5. Повернутися і продовжити знову з кроку №1 (рисунок 1.5)

INCLUDEPICTURE "https://lh6.googleusercontent.com/eWGBd_ka3lnR8EpI7qMtz6CJS5S-ZCBz4inIUDaiCNscCIX-4cI2qm1zJX3HUiGGwcr0J032iN8ncsZYCk24m8KOJGsIzOqX2lCMroAHWVBhIxkIDZ-CogauxYZ8v0e0cg1MNJrw" * MERGEFORMATINET

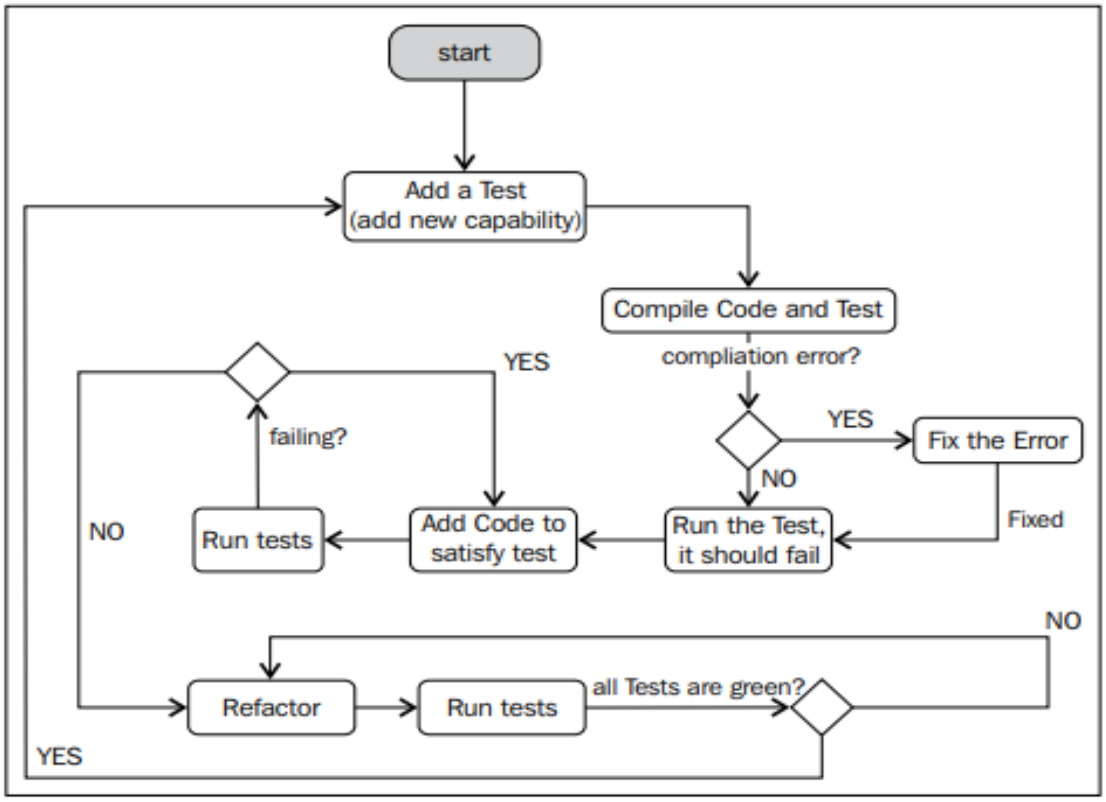


Рисунок 1.5. Схема життєвого циклу TDD

TDD зменшує ринковий час - тобто час, за який платить замовник. З самого початку розробляються невеликі тестові частини; розробники (а інколи тестувальники) можуть перевіряти та приймати функції з самого початку, замість того, щоб чекати завершення розробки.

Наступний рисунок являє собою старий спосіб розробки програмного забезпечення, горизонтальна вісь - це час, а вертикальна вісь є функціями (рисунок 1.6). Тут функції починають рухатися після початку тестування. Таким чином, час виходу на ринок функції залежить від усіх етапів: редизайну, кодування та тестування.

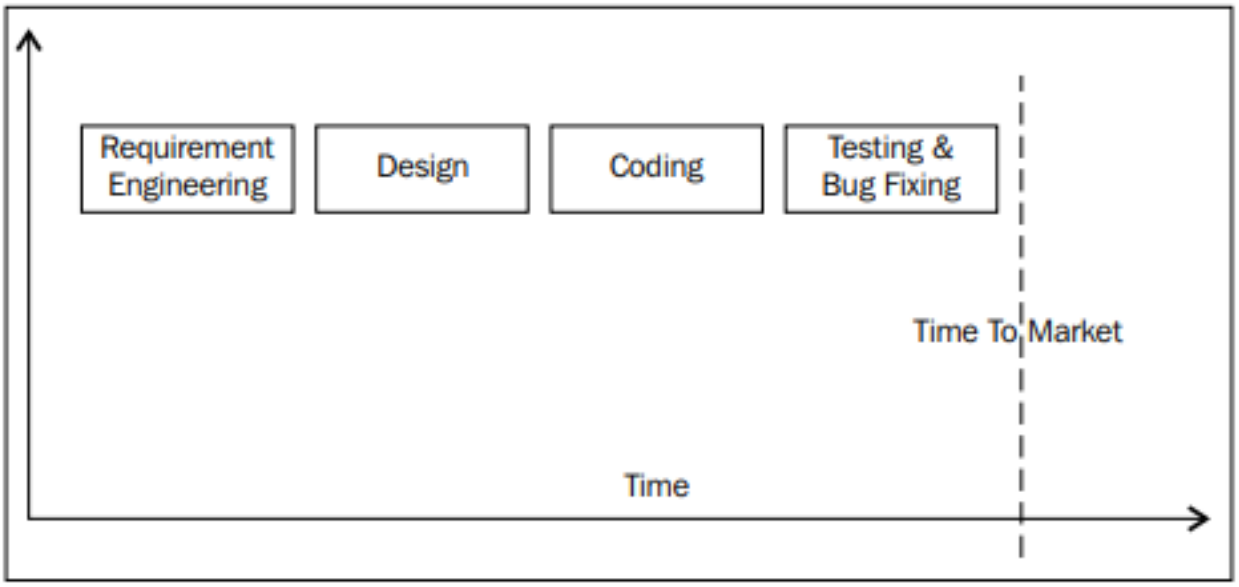


Рисунок 1.6. Модель без TDD підходу

Наступна схема нижче показує життєвий цикл задачі в TDD. Він має ітеративний характер. Тут розробка та тестування починаються дуже рано. Замовлені частини продукту доставляються клієнтам швидше, ніж попередній спосіб. По-перше, проводять вимоги; як тільки будуть зібрані основні вимоги, розпочнеться фаза проектування. Після завершення архітектури базової лінії починається кодування. Нарешті, коли тестована одиниця кодується, починається тестова фаза. Таким чином, функція доставки готова, як тільки буде проведено тестування першого випробуваного модулю.

INCLUDEPICTURE
"https://lh3.googleusercontent.com/rqHom3CFuwjQH3D6vn3UedI-1Gou_mrpec9gCWFvfUipc-tycI2e5AtpHkN2no1Qb0kokJnqrcmfjrr3ePr-x4tBDh5wuykc7tfc9FUjg06et1zNBFQi8I_o2JXx4DwkOxUc6dPM" *
MERGEFORMATINET

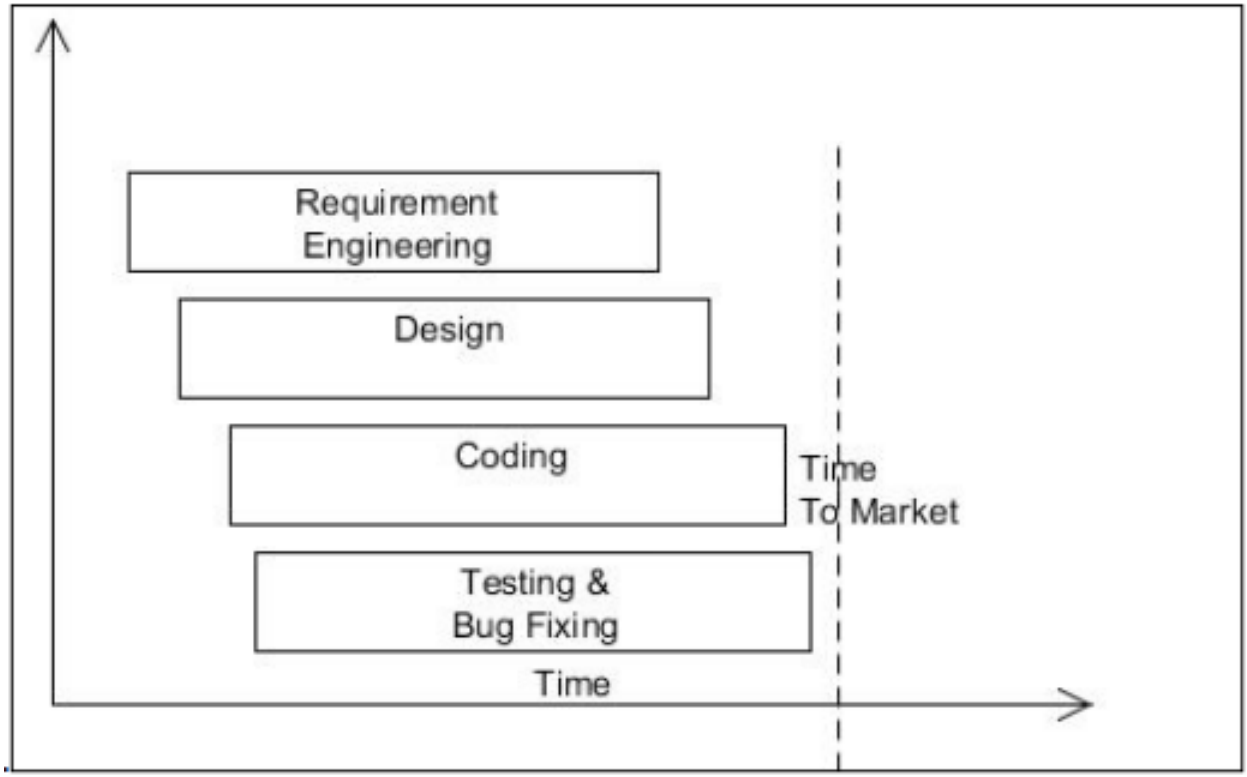


Рисунок 1.7. Модель з використанням TDD

1.3. Бібліотеки, що використовуються для модульного тестування

Іноді неможливо зробити модульне тестування, перевірити компонент коду через недоступність взаємодіючих об'єктів або високу вартість для їх встановлення. Використання тестових класів полегшить цей процес.

Ми знаємо про двійників, що виконують трюки у фільмах - навчені замісники, наймані для небезпечних дій у фільмах, такі як стрибки з будівлі Емпайр Стейт, боротьба на вершині палаючого поїзда, стрибки з літака або подібні дії. Подвійні трюки використовуються для захисту реальних дійових осіб у тому випадку, коли актор недоступний для таких зйомок.

Під час тестування класу, який зв'язується з API, ви не бажаєте викликати API для кожного окремого тесту; наприклад, коли частина коду залежить від

доступу до бази даних, неможливо провести тестування коду, якщо база даних не доступна.

Точно так само під час тестування класу, який підтримує зв'язок із платіжним шлюзом, ви не зможете надсилати платежі на справжній шлюз платежу для виконання тестів.

Єдине, що робить код одиничного тестування настільки важким, - це те, як зовнішні чинники втручаються у процес. Якщо все, що нам доведеться зробити, це написання коду тестів для методів, що сортують масиви або породжують серію Фібоначчі, життя було б легшим. Але в реальному світі ми повинні протестувати код, який використовує бази даних, комунікаційні пристрої, користувацькі інтерфейси та зовнішні програми. Можливо, нам доведеться взаємодіяти з пристроями, які ще не доступні, або моделювати мережеві помилки, які неможливо генерувати локально. Все це змушує зупинити наші одиничні тести на тому, щоб бути акуратними, автономними (і ортогональними) шматочками коду. Натомість, якщо ми не обережні, ми опинимося під час написання тестів, які закінчують ініціалізацію практично всіх компонентів системи, щоб дати тестам досить контекст для запуску. Це не тільки займає багато часу, але і вводиться смішна кількість зчеплення в процесі тестування: хтось змінює інтерфейс або таблицю бази даних, і раптово код установки для вашого бідного тесту на одиницю вимірюється загадковим чином. Навіть найкращі розробники стають невтішними, коли це трапляється кілька разів. Зрештою, тестування починає знижуватися, і ми всі знаємо, де це призводить. На щастя, існує схема тестування, яка може допомогти. Використовуючи дублери, ви можете тестувати код у ізоляції, імітуючи всі ці реальні речі, які в іншому випадку автоматично не зможуть бути протестуватовані. І, як і у багатьох інших практиках тестування, дисципліна використання таких дублюючих об'єктів може покращити структуру коду.

Дублери (англ. Test double) — спеціалізовані методи чи об'єкти, які використовуються при тестуванні систем, в яких виникає необхідність взаємодії з зовнішніми об'єктами, наприклад: бази даних, файлова система, мережеве з'єднання

та тощо. При цьому дублер повинен реалізувати інтерфейс зовнішнього об'єкту та відповідати всім вимогам, які висуваються до реального об'єкта.

Використання дублерів доцільно в наступних випадках:

1. Низька швидкість роботи з зовнішнім об'єктом;
2. Необхідність запуску тестів незалежно від оточення та можливостей машини розробника;
3. Необхідність роботи з реальними та чутливими до змін об'єктами;
4. Складність перевірки коректності взаємодії між частинами;

Та інші випадки, в яких виникає необхідність перевірки стану зовнішнього об'єкту, де це досить складно зробити з зовнішнього коду.

Тестові дублери поділяються на п'ять типів, ниже – про кожен з них.

Фіктивний (dummy) об’єкт: прикладом фіктивних dummy-об’єктів може стати сцена фільму, де двійник не виконує нічого, крім того, що він присутній тільки на екрані. Вони використовуються, коли фактичного актора немає, але їх присутність необхідна для сцени, наприклад, спостереження за тенісним фіналом матчу. Аналогічно, передаються dummy-об'єкти, щоб уникнути NullPointerException для обов'язкових об'єктів, параметрів наступним чином:

```
Book javaBook = new Book("Java 101", "123456");
Member dummyMember = new DummyMember());
javaBook.issueTo(dummyMember);
assertEquals(javaBook.numberOfTimesIssued(),1);
```

У попередньому фрагменті коду був створени dummy-об’єкт і переведений в об'єкт книги, щоб перевірити, чи може книга повідомляти про кількість разів, коли вона була випущена. Тут об'єкт-учасник не використовується ніде, але це потрібно для видання книги.

Заглушка забезпечує непрямі вхідні дані, коли називаються методи заглушки. Вони запрограмовані тільки для тесту. Stubs може записувати іншу інформацію, таку як кількість разів, коли їх викликали, і так далі. Stub піднімає помилку щоразу, коли викликається тестовий метод. Це дозволяє нам перевірити, наприклад,

поведінку транзакції, коли обладнання відсутнє. Mockito дозволяє нам тлумачити інтерфейси та конкретну поведінку. Використовуючи Mockito, ви можете приховати метод dispeng (), щоб виключити виняток.

Підроблені об'єкти (fake) є робочими реалізаціями; в основному, фальшивий клас продовжує оригінальний клас, але зазвичай він має гіршу продуктивність, що робить його непридатним для використання у продакшн. Незважаючи на те, що фіктивні об'єкти є ідеальними інструментами для тестування пристроїв, мок через шаблонні рамки може перетворити ваші модульні тести в такі, що неможливо буде підтримувати.

Основною причиною цієї складності є те, що об'єкти можуть бути занадто великі. Вони мають багато методів, і ці методи повертають інші об'єкти, які також мають методи. Коли ми передаємо макет версії такого об'єкта як параметр, ми повинні переконатися, що всі його методи повертають дійсні об'єкти.

Це призводить до неминучої складності, яка перетворює одиничні тести на “відходи” - їх практично неможливо підтримувати. DAO розширюється від класу Spring та надає API для масового оновлення. Той самий метод використовується для створення нової адреси та оновлення існуючого; якщо кількість не збігається, то виникає помилка. Цей клас не можна перевірити безпосередньо, і йому потрібен getSimpleJdbcTemplate (). Отже, щоб перевірити цей клас, нам потрібно обійти JDBC; ми можемо це зробити, розширивши початковий клас DAO, але перейшовши за допомогою методу сумісного JDBC. Ми можемо використовувати Mockito, щоб створити макет версії JdbcTemplate і повернути його з підробленої реалізації. Цей клас не може бути використаний у виробничому коді, оскільки він використовує макет JdbcTemplate; однак, підроблений клас успадковує всі функціональні можливості DAO, тому це може бути використано для тестування. Фейкові класи дуже корисні для застарілого коду.

Mock (мок-об’єкт**)** - це фіктивний об’єкт**.** Моки об'єктів дотримуються певних умов (очікувань); тест очікує значення з моку об'єкта, а під час виконання мок-об'єкт повертає очікуваний результат. Крім того, мок-об'єкти можуть

відслідковувати калькуляцію, тобто кількість разів, коли метод мок-об'єкту викликається.

Об'єкт Mock використовується для модульного тестування. Якщо у вас є об'єкт, методи якого ви хочете протестувати, і ці методи залежать від будь-якого іншого об'єкта, ви створюєте макет залежності, а не фактичний екземпляр цієї залежності. Це дозволяє вам ізолювати свій об'єкт.

Загальні фреймворки Java для створення mock-об'єктів включають JMock і EasyMock тощо. Зазвичай вони дозволяють створювати mock-об'єкти, поведінку яких ви можете визначити, тому ви точно знаєте, чого очікувати (щодо значень, що повертаються і побічних ефектів) від виклику методів на мок-об'єкті. Як приклад один загальний приклад використання може бути в додатку MVC, де у вас є рівень DAO (об'єкти доступу до даних) і контролер, який виконує бізнес-логіку. Якщо ви хочете, щоб unit test контролер, і контролер, що має залежність від DAO, ви можете зробити макет DAO, який поверне фіктивні об'єкти в ваш контролер.

Важливо відзначити, що зазвичай це випадок, коли макет об'єктів реалізує той же інтерфейс, що і об'єкти, які вони знущаються - це дозволяє вашому коду справлятися з ними через тип інтерфейсу, як якщо б вони були екземплярами реальна річ. Тім Макінтон, Стів Фріман та Філіп Крейг представили концепцію мок-об'єктів у своїй роботі "Ендо-тестування: тестування одиниць з об'єктами". Їх ідея є природним продовженням спеціального ad-hoc стабу, який використовували тестувальники. Різниця полягає в тому, що вони описують структуру, яка дозволяє легше писати мок-об'єкти та об'єднувати їх в модульне тестування.

Їх робота виділяє сім корисних причин для використання мок-об'єкта:

1. Реальний об'єкт має недетерміновану поведінку.
2. Реальний об'єкт важко налаштувати.
3. Реальний об'єкт має поведінку, яку важко викликати (наприклад, мережева помилка).
4. Реальний об'єкт повільний.
5. Реальний об'єкт має (або є) користувальницький інтерфейс.

6. Тесту потрібно запитати реальний об'єкт про те, як його було використано (наприклад, тесту може знадобитися перевірити, чи дійсно було викликано функцію зворотного виклику).
7. Реальний об'єкт ще не існує.

Маккінтон, Фрімен та Крейг також розробили код для структури мок-об'єктів для програмістів Java. Хороша новина полягає в тому, що на додаток до базового коду, пакет може постачатися з багатьма об'єктами, що маскуються на рівні програми. Ви можете знайти фіктивні вихідні об'єкти (OutputStream, PrintStream та PrintWriter), об'єкти, які мокають бібліотеку java.sql, і класи для дублювання середовища. Ми можемо використовувати макети об'єктів двома різними способами. По-перше, ми можемо використовувати їх для створення середовища, в якому працює наш тестовий код: ми можемо ініціалізувати значення в об'єктах, які використовує тестований метод. Як і в інших практиках низького рівня тестування, ви можете виявити, що ваш код стає не тільки кращим у тестуванні, але також краще спроектованим і простішим для розуміння. Мок-об'єкти не вирішать всі ваші проблеми розробки, але вони є надзвичайно потрібними інструментами щоб мати їх у вашому наборі.

Spy (тестовий шпигун) — використовується для тестів взаємодії, головною функцією є запис даних та виклик, що виникають в тестованому об'єкті для подальшої перевірки коректності виклику залежного об'єкту. Дозволяє перевірити логіку саме об'єкту, що тестується, без перевірки залежних об'єктів[4].

Шпигун - це варіація моку/заглушки, але замість того, щоб встановлювати очікування, шпигун записує АПІ-виклики, які надсилаються до об’єкта взаємодії. Наступний приклад пояснює цю концепцію:

```
class ResourceAdapter{
void print(String userId, String document, Object settings) {
if(securityService.canAccess("lanPrinter1", userId)) {
 printer.print(document, settings); }
 }
```

Щоб перевірити поведінку друку класу ResourceAdapter, нам слід знати, чи викликається метод printer.print(), коли користувач має дозволи. При цьому об’єкт взаємодії принтера нічого не робить; він просто використовується для перевірки поведінки ResourceAdapter. SpyPrinter реалізує виклик Printer.print (), збільшує кількість лічильника, який викликається noOfTimes, і getInvocationCount повертає рахунок. Створімо підроблене виконання класу SecurityService для повернення істинного методу canAccess (String printerName, String userId).

Несправжній SecurityService та об'єкти SpyPrinter створюються та передаються класу ResourceAdapter, а потім викликається adapter.print. У свою чергу, очікується, що об'єкт securityService повертає істину, і до нього буде доступ до принтера, а spyPrinter.print (...) збільшить лічильник noOfTimes, що викликається. Нарешті, у попередньому коді ми перевірили, що результатом підрахунку є 1.

Модульний тест - це тест, пов'язаний з єдиною відповідальністю одного класу, який часто називається "System Under Test" (SUT). Метою одиничних тестів є перевірка того, що код працює в SUT. Обстежений об'єкт зазвичай розмовляє з іншими об'єктами, відомими як співавтори. Ці співробітники повинні бути створені таким чином, щоб перевірені об'єкти могли бути присвоєні їм у тесті. Щоб спростити одиничне тестування та дозволити контролювати всі аспекти контексту виконання, корисно замінити реальні об'єкти, що співпрацюють, їх підробленими замінами, які називаються тестовими подвійними. Вони виглядають як оригінали, але не мають жодних залежностей від інших об'єктів. Тестовий подвійність також може бути легко запрограмований з певними очікуваннями, такими як запис будь-яких взаємодій, які вони мали.

Щоб зробити це більш чітким, спробуйте уявити собі код для типової корпоративної системи. Нехай є служба з деякою логікою, яка потребує двох класів для виконання своєї відповідальності. Обидва класи вимагають декількох інших класів. Одним з таких класів може бути DAO, який потребує доступу до бази даних, а інший вимагає черги повідомлень. Було б досить прагнути створити таку ієрархію і забезпечити необхідні ресурси. Там також можуть виникнути проблеми під час

виконання такого типу тесту, наприклад, тривалий час запуску або неможливість одночасного тестування декількох станцій розробника. Однак, використовуючи mocks, той самий тест може бути набагато більш чистим і швидшим. В розробці через тестування (TDD) активно використовуються mock-об'єкти - тип об'єктів, що реалізують задані аспекти модельованого програмного оточення.

Mock-об'єкт являє собою специфічну фіктивну реалізацію інтерфейсу, призначену виключно для тестування взаємодії, і щодо якої висловлюється твердження.

Тобто mocking - це спосіб перевірки функціональності класу в умовах відокремленості. Він не вимагає з'єднання з базою даних або властивостей читання файлів чи файлового сервера для перевірки функціональності. Mock-об'єкти виконують емуляцію справжньої служби.

Mockito - це платформа для тестування з відкритим кодом для Java, випущена в рамках ліцензії MIT. Структура дозволяє створювати тестові об'єкти для розробки на основі тестів (TDD) або керування поведінкою (BDD). Mockito дозволяє макетувати об'єкт створення, перевірка та виправлення. Чому так багато розробників використовують Mockito? Автоматизовані тести - це захист. Вони запускаються та повідомляють користувача, якщо система зламана так що код “порушника” може бути зафіксовано дуже швидко. Якщо тестовий комплект працює протягом години, метою швидкого зворотного зв'язку стає небезпека. Підрозділ тести повинні діяти як захисна мережа та забезпечувати швидкий зворотний зв'язок; це головний принцип від TDD.

Фреймворк Mockito надає ряд можливостей для створення згаданих вище «моків» замість реальних класів або інтерфейсів при написанні JUnit тестів.

Найбільшого поширення набули такі можливості Mockito:

- створення mock-об'єктів для класів та інтерфейсів;

- перевірка виклика методу і значень параметрів переданих методу;

- використання концепції «часткової заглушки», при якій мок створюється на клас з визначенням поведінки, необхідної для деяких методів класу;

- підключення до реального класу «шпигуна» (spy) для контролю виклику методів.

Mockito - це основа для модульних тестів у Java. Вона була розроблена таким чином, щоб бути інтуїтивно зрозумілою для використання, коли потрібні тести. Артефакти Mockito доступні в Maven Central Repository (MCR). Найпростіший спосіб зробити MCR доступним у вашому проекті - це налаштувати наступну конфігурацію у своєму менеджері залежностей. [5]

Власне мок може бути створений за допомогою статичного методу mock (): Flower flowerMock=Mockito.mock(Flower.class);

Але є ще один варіант: використовувати @Mock анотацію:

```
@Mock
  private Flower flowerMock;
```

Проте якщо ви хочете використовувати @Mock або будь-які інші атрибути Mockito, потрібно викликати MockitoAnnotations.initMocks (testClass) або скористатися MockitoJUnit4Runner для запуску в JUnit.

Крім Mockito, використовують також EasyMock, що теж полегшує створення мок-об'єктів. Він використовує Java Reflection для створення «макету» об'єктів для певного інтерфейсу, а це - не що інше, як проксі для реальних реалізацій. Переваги EasyMock:

- немає введення вручну - не потрібно писати моки об'єктів самостійно;

- безпечний рефакторинг - перейменування імен методу інтерфейсу або параметри реорганізації не порушать тестовий код, оскільки мок-об’єкти створюються під час виконання;

- підтримка повернення значень;

- підтримка перевірки перевірки порядку викликів методу;

- підтримка анотацій[6].

Якісний код неможливий без тестів. А якісні тести - без моків. У створенні моків нам давно допомагають різні корисні бібліотечки, на зразок EasyMock або Mockito. Але, на жаль, Mockito не став “срібною кулею”. Обмеженням завжди були final класи, private поля і методи, static методи і багато іншого. І доводилося вибирати: або красивий дизайн, або якісне покриття тестами. Альтернативою є така бібліотека як PowerMock. Розглянемо його можливості далі.

Нехай в нашому коді використовується final клас, виклик методу якого нам необхідно перевірити. Mockito тут безсилий, у цього класу немає інтерфейсу, а сам клас не може мати спадкоємців. Що-небудь змінити ми теж не можемо - або в силу архітектурних особливостей, або в силу того, що це сторонній сервіс. Перше, на що кидається погляд - анотації @RunWith & @PrepareForTest. Перша необхідна, щоб замінити стандартний JUnit виконавець тестів на PowerMock, який використовує дію класслоадера, що б вирішити проблему створення mock-об'єкта з final класу. Друга анотація підказує виконавцю тесту, які класи необхідно підготувати для тесту. Для створення mock-об'єкта ми використовуємо метод з набору PowerMockito.
Ще одна проста і цікава можливість - перевіряти виклики static методів. Анотації @RunWith & @PrepareForTest так само необхідні для роботи з static методами. Розглянемо, для чого необхідні нові інструкції:

PowerMockito.mockStatic (Class <?> Type) - створює mock для всіх статики методів в заданому класі. Варто відзначити, що можна створити mock тільки для необхідних методів.

PowerMockito.when (T methodCall) .thenReturn (returnValue) - стандартний спосіб задати якесь поведінку створеної заглушки.

PowerMockito.verifyStatic () - викликається перед перевіркою кожного статичного виклику методу.

ExternalMegaService.doStatic () - визначає, який власне метод повинен був бути викликаний.

Ще одна чудова можливість PowerMock'a - mock'ати створення нових об'єктів.

Конструкція PowerMockito.whenNew (Class <?> Type) .withNoArguments (). ThenReturn (instance) говорить PowerMock'у замінити в інспектованих класах створення об'єктів типу type на об'єкт instance. Важливо, щоб об'єкт, в якому необхідно замінити створення mock об'єкта, створювався після цієї конструкції. Так само слід зазначити, що ExternalServiceFactory може бути непростим об'єктом, а partial mock'ом (spy) і тоді його поведінку теж можна буде перевірити.

Ложкою дьогтю є те, що якщо вам необхідно проінструктувати клас (@PrepareForTest), який ви тестуєте, то ви ніколи не дізнаєтеся ступінь покриття даного класу тестами, тому що coverage тул не зможе його проінспектувати. У таких випадках можна поділити тест на два класи. У першому перевірити все, що можна перевірити без інструктування тестованого класу, у другому - тільки те, для чого необхідно робити @PrepareForTest. Ось такі можливості для тестування надає PowerMock.

JUnit - найпопулярніша бібліотека для юніт-тестування у мові Java. Він надає певні методи підтвердження для всіх примітивних типів і об'єктів і масивів (примітивів або об'єктів). Порядок параметрів - очікувана величина, а потім - фактична величина. За бажанням першого параметра може бути повідомлення String, яке виводиться при несправності. Існує дещо інше твердження, assertThat що має параметри додаткового повідомлення про помилку, фактичне значення та Matcher об'єкта. При цьому, очікувані та фактичні значення відхиляються в порівнянні з іншими методами твердження. Дослідницьке опитування, проведене в 2013 р. на
10 000 проектів Java, розміщених на GitHub, виявило, що JUnit (у поєднанні з slf4j-api) була найчастіше включеною зовнішньою бібліотекою. Кожна бібліотека була використана 30,7% проектів.

JUnit - бібліотека для модульного тестування програмного забезпечення, створена Кентом Беком і Еріком Гаммою, вона належить до сімейства фреймворків xUnit для різних програмних мов, що беруть початок в SUnit Кента Бека для Smalltalk.

Висновки до розділу 1

Тестування програмного забезпечення має бути не лише корисним, але й максимально раціональним. На сьогоднішній день практично у кожному прооекті, що прагне максимальної якості, розробляються юніт-тести. Вони запускаються щоразу разом зі збіркою спочатку на машині розробника, що вносить зміни, щоб провалідувати код, перш ніж він потрапить далі. Потім знову йде перевірка - на сервері. Все це займає час.

Для модульних тестів на Java найчастіше використовують бібліотеку JUnit. Близько року назад вийшла 5-та версія з багатьма перевагами, які спонукають розробникив переходити на неї зі старіших версій. Проте за замовчуванням тут забезпечено виконання модульних тестів лише послідовно. А отже, не вистачає функціоналу, який би прискорив час на виконання тестів, запустивши їх у паралельному режимі.

2 ОПИС АЛГОРИТМІВ

2.1 Поняття багатопоточності

На відміну від багатьох інших комп'ютерних мов, Java надає вбудовану підтримку для багатопотокового режиму. Багатопоточність в Java містить дві або більше частин, які можуть працювати одночасно. Потік Java насправді є легким процесом.

Кожна частина такої програми називається потоком, і кожен потік визначає окремий шлях виконання. Таким чином, багатопоточність - це спеціалізована форма багатозадачності.

Система керування часом Java залежить від потоку для багатьох речей. Потоки зменшують неефективність, запобігаючи витрачання циклів процесора.

Потоки існують у кількох станах. Нижче наведені ці стани:

- Новий (New)- Коли ми створюємо примірник класу Thread, потік знаходиться в новому стані.

- В процесі (Runnable) - Java потік знаходиться у стані експлуатації.

- Призупинено (Suspended) – рпрацюючий поток може бути призупинено, він тимчасово припиняє свою діяльність. «Підвішений» потік потім можна відновити, дозволяючи йому повернутися на місце, де він був.

- Заблоковано (Blocked) – роботу потоку Java можна заблокувати, коли чекає на ресурс.

- Припинено (Terminated) - потік можна припинити, що припиняє його виконання негайно в будь-який момент часу. Після завершення потоку його не можна відновити. Отже, мова йшла про стан Java Thread. Тепер давайте перейдемо до найважливішої теми потоків Java, тобто класу потоку та інтерфейсу, що запускається.

Є два способи створення потоку в Java:

1) Розширюючи клас Thread.

2) Використовуючи інтерфейс Runnable.

У класа Thread існують наступні методи:

- getName(): Він використовується для отримання назви потоку
- getPriority(): Отримайте пріоритет потоку
- isAlive(): Визначте, чи нитка все ще працює
- join(): Зачекайте, поки нитка закінчиться
- run(): Точка входу для потоку
- sleep(): призупинити потік протягом певного періоду часу
- start(): запустити потік, викликавши його метод run ()

Пріоритетами потоку є цілі числа, які вирішують, як слід розглядати один потік по відношенню до інших. Пріоритет потоку вирішує, коли перемикатися від одного потоку до іншого, процес називається перемиканням контексту. Потік може добровільно відпустити контроль, а найвищий пріоритет потоку, який готовий до запуску, надається ЦП. Потік може бути витіснений потоком з вищим пріоритетом, незалежно від того, що робить потік з нижчим пріоритетом. Кожного разу, коли гілка старшого пріоритету бажає запуститися, саме так і відбувається. Для встановлення пріоритету використовується setPriority() - метод потоку , який є методом класу ThreadClass.

Замість визначення пріоритету у цілих числах ми можемо використовувати MIN_PRIORITY, NORM_PRIORITY або MAX_PRIORITY.

У всіх практичних ситуаціях основний потік повинна закінчуватися, інакше закінчуються і інші потоки, які виросли з основного потоку. Щоб дізнатись, чи закінчився потік, ми можемо викликати isAlive(), яка повертає true, якщо поток активний. Інший спосіб досягти цього - за допомогою join() методу - цей метод, коли він викликається з батьківского потоку, робить так що батьківський потік очікує, доки не закінчиться підпотік(child thread). Ці методи визначаються в Thread класі.

Багатопотоковий процес вводить асинхронну поведінку до програм. Якщо потік пише деякі дані, інший потік може читати ті самі дані в той же час. Це може призвести до непослідовності. Коли два або більше потоків потребують доступу до спільного ресурсу, має бути певний спосіб, коли ресурс буде використовуватися лише одним ресурсом за один раз. Процес досягнення цього називається синхронізацією.

Для реалізації синхронної поведінки Java має синхронний метод. Коли потік знаходиться всередині синхронізованого методу, жоден інший потік не може викликати жодного іншого синхронізованого методу на одному і тому ж об'єкті. Потім всі інші потоки чекають, поки перша нитка вийде з синхронізованого блоку. Коли ми хочемо синхронізувати доступ до об'єктів класу, який не був розроблений для багатопотокового доступу, і код методу, який має бути доступним синхронно, недоступний у нас, у цьому випадку ми не можемо додати синхронізовані з відповідними методами. У java ми маємо рішення для цього, поставимо виклики методам (які потрібно синхронізувати), які визначаються цим класом усередині синхронізованого блоку наступним чином.

```
Synchronized(object){
  // statement to be synchronized
}
```

Існує декілька методів, за допомогою яких потоки можуть спілкуватися один з одним. Ці методи, наприклад, wait(), notify(), notifyAll(). Всі ці методи можна викликати лише з синхронізованого методу.

Для розуміння синхронізації, припустимо, Java має концепцію монітора. Монітор може розглядатися як коробка, яка може містити лише один потік. Коли потік потрапляє на монітор, всі інші потоки повинні зачекати, доки цей потік не вийде з монітора.

Далі wait() повідомляє викликаному потоку, щоб відмовитися від монітора та перейти до сну, доки інший потік не потрапить до одного монітора та виклич е notify().

Через notify() активується перший потік, який викликає wait() на тому самому об'єкті.

А notifyAll() пробуджує всі потоки, які викликалися wait () на одному і тому ж об'єкті. Потік з найвищим пріоритетом запуститься спочатку - найпершим.

2.2. Механізм рефлексії Java

Java Reflection - рефлексія - дає змогу перевіряти класи, інтерфейси, поля та методи під час виконання, не знаючи назви класів, методів тощо під час компіляції. Також можна ініціалізовувати нові об'єкти, викликати методи та отримувати / встановлювати значення поля за допомогою рефлексії.

Рефлексія Java досить потужна і може бути дуже корисною. Наприклад, Java Reflection може бути використаний для відображення властивостей файлів JSON до методів getter / setter у об'єктах Java, таких як Jackson, GSON, Boon та ін . Або ще рефлексію можна використовувати для позначення назв стовпців JDBC ResultSet для методів getter / setter у об'єкті Java.

Ось приклад на Java, який покаже вам, що виглядає як рефлексія:

```
Method[] methods = MyObject.class.getMethods();
for(Method method : methods){
   System.out.println("method = " + method.getName());
}
```

Цей приклад отримує Classоб'єкт з класу, який викликається MyObject. Використовуючи об'єкт класу, приклад отримує список методів у цьому класі, повторює методи та друкує їхні імена.

Використовуючи рефлексію в Java, відправною точкою часто є Class об'єкт, який представляє певний клас Java, який ви хочете перевірити через рефлексію. Наприклад, щоб отримати Class об'єкт для класу з іменем MyObject, можна написати:

```
Class myObjectClass = MyObject.class;
```

Таким чином отримаємо посилання на Class об'єкт для MyObject класу.

Коли ви маєте посилання на Class об'єкт, що представляє якийсь клас, ви можете побачити, які поля містить цей клас. Ось приклад доступу до полів класу Java:

```
Class myObjectClass = MyObject.class;
Field[] fields    = myObjectClass.getFields();
```

З посиланням на Field екземпляр рефлексії Java ви можете почати перевірку поля. Ви можете ознайомитися з його іменем, модифікаторами доступу тощо.

Використовуючи Java Reflection можна дізнатись, які конструктори даного класу Java і які параметри вони беруть і т.д.

Ви також можете побачити, які методи даного класу від свого Class об'єкта. Ось приклад доступу до методів даного класу за допомогою відображення Java:

```
Class myObjectClass = MyObject.class;
Method[] methods = myObjectClass.getMethods();
```

Після того як ви маєте посилання на Methodекземпляр відображення Java, ви можете почати перевірку. Ви можете прочитати назву методу, які параметри він приймає, тип повернення і т. д.

Ви також можете використовувати рефлексію Java, щоб з'ясувати, які методи отримання та встановлення класу мають місце.

Ви навіть можете отримати доступ до приватних полів і методів через рефлексію Java - навіть з-за меж класу, що володіє приватним полем або методом.

Крім того, можна використовувати Java Reflection для інтроспективності Java-масивів. Наприклад, ви можете визначити, який тип класу масив масиву. Наприклад, якщо ви переглядаєте масив рядка, ви можете визначити, що тип елемента - String, перевіряючи клас масиву.

Можна проаналізувати загальні типи полів, параметри методу та параметри повернення методу, якщо вони оголошуються загальним типом.

Відображення Java має спеціальний Proxy клас, який може реалізувати інтерфейс Java динамічно під час виконання, а не під час компіляції. Для

динамічного проксі надається об'єкт обробника, який перехоплює всі виклики методу на динамічному проксі. Це може бути дуже зручним способом вирішення деяких типів завдань, як додавання керування транзакцією за допомогою викликів методів, реєстрації журналів або інших типів бажаної поведінки.

У Java можна динамічно завантажувати, а також перезавантажувати класи за допомогою Java ClassLoader. ClassLoader клас насправді не є частина Reflection API Java, але Java Reflection часто використовуються для досягнення «динамічної» поведінки (поведінкові змін під час виконання) і динамічного завантаження класів.

Деякі анотації Java ще доступні під час виконання. Якщо клас Java має анотації, доступні під час виконання, ви також можете отримати доступ до них за допомогою відображення Java.

Отже за допомогою Java Reflection можна отримати доступ до анотацій, доданих до класів Java, під час виконання.

Анотації - це свого роду коментар або метадані, які ви можете вставити у ваш код Java. Ці анотації потім можуть оброблятися під час компіляції інструментами попереднього компілятора або під час виконання через Java Reflection. Ось приклад анотації класу:

```
@ MyAnnotation (name = "someName", value = "Hello World")
publicclass TheClass {
}
```

Клас TheClass має анотацію, @MyAnnotation написану зверху. Анотації визначаються як інтерфейси. Ось визначення для MyAnnotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
   public String name();
   public String value();
}
```

Знак “@” перед інтерфейсом позначає його як анотацію. Після визначення анотації ви можете використовувати його у своєму коді далі.

Дві директиви у визначенні анотації, @Retention(RetentionPolicy.RUNTIME), а також @Target(ElementType.TYPE) вказують, як використовувати анотацію.

@Retention(RetentionPolicy.RUNTIME) означає, що анотацію можна отримати за допомогою відображення під час виконання. Якщо ви не встановите цю директиву, анотація не буде збережена під час виконання, і тому не доступна через відображення.

@Target(ElementType.TYPE)означає, що анотацію можна використовувати лише на типах (класи та інтерфейси зазвичай). Ви також можете вказати METHODабо FIELD, або ви можете залишити цільове слово взагалі, так що анотацію можна використовувати як для класів, так і для методів і полів.

Ви можете отримати доступ до анотацій класу, методу або поля під час виконання. Можна також додати анотації до декларацій параметрів методу, наприклад, як @MyAnnotation(name="aName", value="aValue") String parameter)

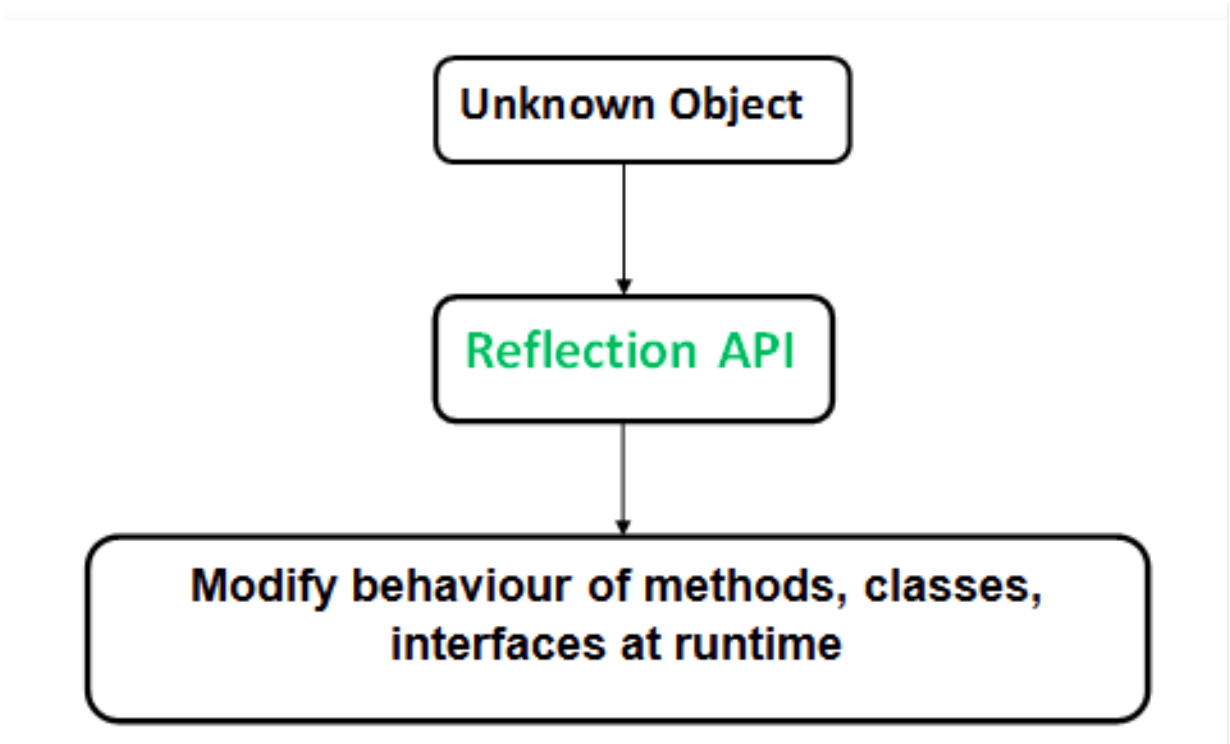


Рисунок 2.1. Схема дії рефлексії

Є у рефлексії певні недоліки:

- Накладні витрати на продуктивність: рефлективні операції мають меншу продуктивність, ніж їх невідбиваючі аналоги, і їх слід уникати в розділах коду, які часто називаються в продуктах-чутливих додатках

- Вплив внутрішніх елементів: відображуваний код порушує абстракції, тому може змінювати поведінку з оновленнями платформи.

Проте й переваги використання рефлексії є очевидними:

- Особливості розширення: програма може використовувати зовнішні, визначені користувачем класи, створюючи екземпляри об'єктів розширюваності з використанням їх повноцінних імен.

- Інструменти налагодження та тестування : Debuggers використовують властивість рефлексії для вивчення приватних членів класів.

2.3. Застосування Fork Join

Fork join - метод, застосовуваний в комунікаційних і комп'ютерних системах: що слугує для збільшення продуктивності виконання великої кількості робочих завдань . Метод полягає в тому, що кожна задача розбивається на безліч дрібніших синхронізованих завдань, які обробляються паралельно на різних серверах.

Суть методу проста: велике завдання розбивається на завдання поменше, ті, в свою чергу, на ще більш дрібні завдання, і так до тих пір, поки це має сенс. В самому кінці тривіальна задача, яка вийшла, виконується послідовно. Даний етап називається Fork. Результат виконання послідовних завдань об'єднується вгору по

ланцюжку, поки не вийде рішення самої верхньої завдання. Даний етап називається Join. Виконання всіх завдань відбувається паралельно[8].

Для програм, для яких потрібні окремі або спеціальні пули (хранилища для збереження) {@code ForkJoinPool}, може бути побудований з заданим рівнем паралелізму цілі (рисунок 2.2); за замовчуванням, дорівнює кількості доступних процесорів. Пул намагається зберегти достатньо активних (або доступних) потоків, динамічно додаючи, призупиняючи або відновлюючи внутрішні робочі теми, навіть якщо деякі завдання затримуються, очікуючи приєднання до інших.

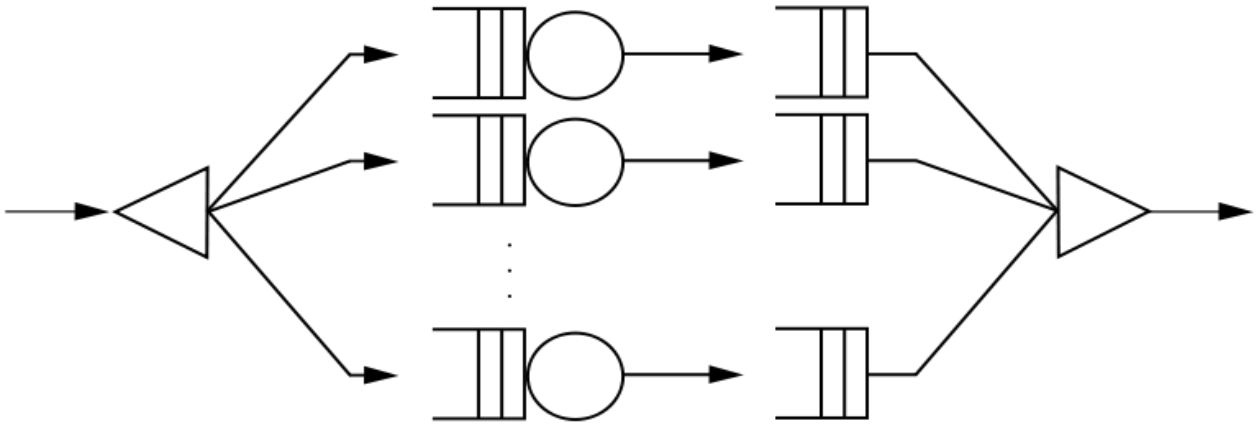


Рисунок 2.2. Схематичне зображення методу Fork Join

Проте жодні такі коригування не гарантуються в умовах заблокованих I / O або інших некерованих синхронізацій. Вкладений інтерфейс {@link ManagedBlocker} дозволяє розширювати види збереженої синхронізації. Політика за умовчанням може бути скасована за допомогою конструктора з параметрами, відповідними документам у класі {@link ThreadPoolExecutor}.

На додаток до методів управління виконанням та життєвим циклом, цей клас надає методи перевірки стану (наприклад, {@link #getStealCount}), які призначені для розробки, настроювання та моніторингу додатків fork/ join. Крім того, метод {@link #toString} повертає показники стану пула в зручній формі для неформального моніторингу. Як і у випадку з іншими ExecutorService's, існує три основні виконання завдання. Вони призначені для використання в першу чергу клієнтами, які ще не займаються обчисленнями вилучення/приєднання в поточному

пулі. Основні форми цих методів приймають екземпляри {@code ForkJoinTask}, але перевантажені форми також дозволяють змішане виконання звичайних {@code Runnable} або {@code Callable} заходів. Проте завдання, які вже виконуються в пулі, замість того, щоб замість використання асинхронізованих завдань у стилі події, які зазвичай не приєднуються, зазвичай використовують формати, що входять до обчислення, перераховані в таблиці, в цьому випадку вибір методів незначний.

Цей клас і його вкладені класи забезпечують головне функціональність та управління для набору робочих потоків: матеріали з потоків не-FJ вводяться в черги для подання.Виконавці (Workers) виконують ці завдання і, як правило, поділяють їх на підзадачі, які можуть бути використаними іншими виконувачами. Робота на основі рандомного сканування зазвичай призводить до кращої пропускної спроможності, ніж "Робота", в якій виробники призначають завдання для простою потоків, частково тому, що теми, які раніше виконували інші завдання сигнал про потік прокидається (це може бути тривалий час) може замість цього візьміть це завдання. Правила переваг надають першочергове значення - обробка завдань з власних черг (LIFO або FIFO, залежно від режиму), та до рандомних FIFO бере завдання в іншій черги. Ця структура почалася як засіб підтримки дерево-структурований паралелізм з використанням відбирання роботи. Через деякий час, його переваги масштабованості призвели до розширення та зміни до краще підтримувати більш різноманітні контексти використання. Тому що більшість внутрішніх методів та вкладених класів взаємопов'язані, їх основне обґрунтування та описи представлені тут; індивідуальні методи і вкладені класи містять лише короткі коментарі та подробиці.

Будь-який з декількох дій може бути зроблений, коли один працівник чекає приєднання до завдання, викраденого (або завжди тримається) іншим. Оскільки ми мультиплексуємо багато завдань в пул працівників, ми не можемо просто заблокувати їх (як у Thread.join). Ми також не можемо просто перепризначити стека часу виконання столяра іншим і замінити його пізніше, що буде формою "продовження", що навіть, якщо це можливо, не обов'язково є гарною ідеєю,

оскільки нам може знадобитися як розблоковане завдання, так і його продовження прогрес

Замість цього ми об'єднаємо дві тактики:

- Допомога: організувати виконання певного завдання, яке він буде опрацьовувати, якщо не міг забрати на себе

- Компенсація: якщо існує достатньо живих потоків, метод tryCompensate () може створити або повторно активувати запасний потік, щоб компенсувати заблокованих, поки вони не розблокуються.

Третя форма (виконана в tryRemoveAndExec) полягає в тому, щоб допомогти гіпотетичному компенсатору: якщо ми можемо легко сказати, що можлива дія компенсатора полягає в тому, щоб забрати на себе і виконати об'єднання завдання, приєднання потоку може зробити це безпосередньо, без необхідності Компенсаційного потоку. API розширень ManagedBlocker не може використовувати допомогу, тому покладається лише на компенсацію методом awaitBlocker.

Алгоритм у awaitJoin передбачає форму "лінійної допомоги". Кожен воркер записує ідентифікатор черги, з якої він в минулому вкрав завдання. Спосіб сканування в очікуванні джойна використовує ці маркери, щоб спробувати знайти воркера, який допомагає (тобто забрати на себе завдання та виконувати його), що могло прискорити завершення активно об'єднаного завдання. Таким чином, воркер виконує завдання, яке було б на власній локації, якщо завдання, що підлягає об'єднанню, не було «викрадено». Це консервативний варіант підходу, описаного в Wagner & Calder "Leapfrogging: портативна техніка для впровадження ефективних ф'ючерсів" SIGPLAN Notices, 1993. Це відрізняється головним чином тим, що ми записуємо лише ідентифікатори черги, а не повні посилання належності. Для цього потрібне лінійне сканування масиву workQueues для виявлення стілерів(stealers – «викрадачі»), але виділяє вартість, коли це потрібно, а не додавання накладних витрат на завдання. Пошуки можуть не вдаватися в пошуку стілерів GC об‘єктів і подібних джерел записування затримки. Крім того, навіть якщо вони будуть точно визначені, "викрадачі" (stealers) можуть ніколи не спродукувати задачу, допомогти

із завданням, в якій би могло допомогти об’єднання (join). Отже, компенсація провадиться після неможливості пошуку завдань.

Компенсація за замовчуванням не спрямована на те, щоб у будь-який момент часу зберігати точно цільову паралельність з розблокованими потоками. Деякі попередні версії цього класу використовували негайні компенсації за будь-яке заблоковане об'єднання. Проте на практиці переважна більшість блокування є тимчасовими побічними продуктами GC та іншими діями JVM або OS, які погіршуються шляхом заміни, якщо вони викликають довгострокове надмірне надходження. Замість того, щоб запроваджувати довільну політику, ми дозволяємо користувачам перевизначати за замовчуванням лише додавання потоків до явного голоду. Механізм компенсації також може бути обмежений. Межі для commonPool краще дозволяють JVMs справлятися з помилками та зловмисниками програмного забезпечення, перш ніж вичерпати ресурси, щоб це зробити.

У стаціонарному режимі існує суворе (деревно-структуроване) обчислення, кожен потік робиться доступним для відбирання на себе достатніх завдань для інших потоків, щоб залишатися активними. Відповідно, якщо всі потоки відтворюються за тими ж правилами, кожен потік повинен зробити доступним лише постійну кількість завдань.

Мінімальна корисна константа - лише 1. Але, використовуючи значення 1, потрібно буде негайно поповнити кожну крадіжку, щоб підтримувати достатньо завдань, що є нездійсненним. Крім того, розбиття / деталізація запропонованих завдань має звести до мінімуму швидкість викрадення, що в загальному означає, що потоки ближче до дерева обчислень повинні генерувати більше, ніж ті, що ближче до нижньої. У ідеальному стаціонарному стані кожна потік на приблизно однаковому рівні дерева обчислень. Однак виробництво додаткових завдань амортизує невизначеність прогресу та припущень дифузії.

Таким чином, користувачі хочуть використовувати значення, більші (але не набагато більші) ніж 1, для того, щоб одночасно здійснювати плавний перехідний дефіцит та захистити від нерівномірного прогресу; оскільки обмінювалися на

вартість додаткових витрат на накладні витрати. Користувач вибирає граничне значення для порівняння з результатами цього дзвінка, щоб керувати рішеннями, але рекомендовано такі значення, як 3.

Коли всі потоки активні, це в середньому нормально, щоб оцінити надлишок строго локально. У стаціонарному стані, якщо підтримується одна гілка, то можна додати 2 додаткових завдання. Отже, ми можемо просто використовувати оцінювану довжину черги. Проте лише ця стратегія призводить до серйозних помилкових оцінок в деяких нестаціонарних умовах (нахил, падіння, інші). Ми можемо виявити багато з них шляхом подальшого вивчення кількості «холостих» потоків, які, як відомо, мають нульові черзі завдань, тому компенсувати фактор (#idle/ #active) потоків.

Висновки до розділу 2

Беззаперечним плюсом багатопоточності є можливість запускати потоки у паралельному режимі, скорочуючи тим самим час на їх виконання. Це можна використати, у тому числі, і у застосуванні для реалізації засобів для модульного тестування.

Також я застосувала fork join - метод, що слугує для збільшення продуктивності виконання великої кількості робочих завдань і полягає в тому, що кожна задача розбивається на безліч дрібніших синхронізованих завдань, які обробляються паралельно на різних серверах. І оскільки метою є скорочення часу на виконання – то саме цей метод став корисним.

Крім того, у моїй роботі був використаний механізм рефлексії, адже використання анотацій стало однією із ключових особливостей пропонованого рішення. А зчитування анотацій без цього механізму практично неможливе.

3 ВИБІР ТА ОПИС ЗАСОБІВ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1. Середовище розробки IntelliJ IDEA

Для розробки додатку було використано платформу IntelliJ IDEA - це інтегроване середовище розробки Java (IDE) для розробки комп'ютерного програмного забезпечення. Редактор IntelliJ IDEA є особливим. Найбільш помітним є те, що ви можете запускати практично будь-яку функцію IDE, не залишаючи її, що дозволяє вам організувати макет, де ви маєте більше місця на екрані, оскільки приховані допоміжні елементи керування, такі як панелі інструментів та вікна .

IntelliJ IDEA підтримує пробні тести на платформі JUnit з версії 2016.2. Однак слід зазначити, що рекомендується використовувати IDEA 2017.3 або новішу версію, так як ці нові версії IDEA буде завантажити наступні JARs автоматично на основі версії API , що використовується в проекті: junit-platform-launcher, junit-jupiter-engine і junit-vintage-engine.

IntelliJ IDEA випускає до випуску специфічних версій JUnit 5 для пакету IDEA 2017.3. Отже, якщо ви хочете використовувати нову версію JUnit Jupiter, виконання випробувань в середовищі IDE може виникнути з-за конфліктів версій. У таких випадках, будь ласка, дотримуйтесь наведених нижче інструкцій, щоб використовувати нову версію JUnit 5, ніж таку, що входить до комплекту IntelliJ IDEA.

Для того , щоб використовувати іншу версію JUnit 5 (наприклад, 5.3.2), буде потрібно включити відповідні версії junit-platform-launcher, junit-jupiter-enginei junit-vintage-engine JARs в шляху до класів.[7]

Розробка великих програмних продуктів майже неможлива без застосування спеціальних інтегрованих середовищ розробки (IDE), що значно спрощують роботу з проектом. Окрім можливостей звичайного текстового редактору, IDE спрощують роботу з системами контролю версій, комунікацію з базою даних, інтеграцію з

фреймворками та інструментами для збірки та тестування систем. Для Java існує три основні IDE: IntelliJ IDEA, NetBeans та Eclipse. NetBeans та Eclipse є безкоштовними та підтримують майже всі основні фреймворки та бібліотеки. Середовище IntelliJ IDEA(розширене) є платним та, окрім базових можливостей інших IDE, є більш стабільним, має кращі механізми пошуку у проекті, широкий набір додаткових якісних плагінів та зручні методи для рефакторингу коду. Для студентів є можливість отримати IntelliJ IDEA абсолютно безкоштовно.

Саме тому весь код був написаний у інтегрованому середовище розробки IntelliJ IDEA для різних мов програмування від компанії JetBrains.

До основних можливостей IntelliJ IDEA відносяться:

— розумне автодоповнення, інструменти для аналізу якості коду, зручна навігація, розширені рефакторингом і форматування для Java, Groovy, Scala, HTML, CSS, JavaScript, CoffeeScript, ActionScript, LESS, XML і багатьох інших мов;

— підтримка всіх популярних фреймворків і платформ, включаючи Java EE, Spring Framework, Grails, Play Framework, GWT, Struts, Node.js, AngularJS, Android, Flex, AIR Mobile і багатьох інших;

— інтеграція з серверами додатків, включаючи Tomcat, TomEE, GlassFish, JBoss, WebLogic, WebSphere, Geronimo, Resin, Jetty і Virgo;

— інструменти для роботи з базами даних і SQL файлами, включаючи зручний клієнт і редактор для схеми бази даних;

— інтеграція з комерційними системами управління версіями Perforce, Team Foundation Server, ClearCase, Visual SourceSafe;

— інструменти для запуску тестів і аналізу покриття коду, включаючи підтримку всіх популярних фреймворків для тестування.

До складу IntelliJ IDEA входить модуль візуального проектування програм GUI-інтерфейсу Swing UI Designer, XML-редактор, редактор регулярних виразів, система перевірки коректності коду, система контролю за виконанням завдань і доповнення для імпорту та експорту проектів з Eclipse. Доступні засоби інтеграції з системами відстеження помилок JIRA, Trac, Redmine, Pivotal Tracker, GitHub,

YouTrack, Lighthouse. Під час розробки даного програмного забезпечення було використано 2018.1 версію продукту.

3.2. Мова програмування Java

Основною мовою для розроблення додатку було обрано Java. Об'єктно-орієнтована мова Java, розроблена в компанії Sun Microsystems в 1995 році для відображення графіки на стороні клієнта за допомогою аплетів, в даний час використовується для створення незалежних від операційних систем програм.

Мова Java знайшла широке застосування в Інтернет-додатках, додавши на статичні і клієнтські веб-сторінки динамічну графіку, поліпшивши інтерфейси і реалізувавши обчислювальні можливості. Але об'єктно-орієнтована парадигма і незалежність від операційної системи привели до того, що вже буквально через кілька років після створення мову практично повністю покинула клієнтські сторінки і перебралась на сервери. На стороні клієнта його місце зайняли мови JavaScript, Adobe Flash і ін.При створенні мови Java, розробники намагалися зробити її більш простою, ніж її предок C ++. Сьогодні з появою нових версій можливості мови Java істотно розширилися і багато в чому перекривають функціональність C ++. Java вже не поступається за складністю попередникам і називати її простою мовою неможливо.

Відсутність вказівників (найбільш небезпечного засобу мови C ++) не можна вважати звуженням можливостей, а тим більше - недоліком, це просто потреба безпеки. Можливість роботи з довільними адресами пам'яті через ігнорування типів дозволяє ігнорувати захист пам'яті. Відсутність в Java множинного успадкування легко замінюється на більш зрозумілі конструкції із застосуванням інтерфейсів.

Системна бібліотека класів мови Java містить класи і пакети, реалізуючи і розширюючи базові можливості мови, мережева взаємодія, взаємодія з базами даних, графічні інтерфейси і багато іншого. Методи класів, включених в ці бібліотеки, викликаються JVM (Java Virtual Machine) під час інтерпретації програми.

У Java всі об’єкти програми розташовані в динамічній пам’яті - купі даних (heap) і доступні за об’єктним посиланнях, які, в свою чергу, зберігаються в стеку (stack). Це рішення виключило безпосередній доступ до пам’яті, але ускладнило роботу з елементами масивів і зробило її менш ефективною в порівнянні з програмами на C ++. У свою чергу, в Java запропоновано удосконалений механізм роботи з колекціями, реалізовані основні динамічні структури даних. Необхідно відзначити, що об’єктна посилання мови Java містять інформацію про клас об’єкту, на котрий вона посилається, так що об’єкт на посилання - це не покажчик, а дескриптор (опис) об’єкта. Наявність дескрипторів дозволяє JVM виконувати перевірку сумісності типів на фазі інтерпретації коду, генеруючи виключення в разі помилки. В Java змінена концепція організації динамічного розподілення пам’яті: відсутні способи програмного очищення динамічно виділеної пам’яті.

Замість цього реалізована система автоматичного звільнення пам’яті (збирач сміття), виділеної за допомогою оператора new. Програміст може тільки рекомендувати системі звільнити виділену динамічну пам’ять.

На відміну від C ++, Java не підтримує множинне успадкування, перегрузку операторів, беззнакові цілі, пряме індексування пам’яті і, як наслідок, покажчики.

В Java існують конструктори, але відсутні деструктори (застосовується автоматичне прибирання сміття), не використовується оператор goto і слово const, хоча вони є зарезервованими словами мови.

3.3 Бібліотека JUnit 5

JUnit 5 - це наступне покоління JUnit. Метою є створення новітньої основи тестування на JVM на стороні розробників. Це включає в себе зосередження уваги на Java 8 і вище, а також увімкнення багатьох різних стилів тестування.

На відміну від попередніх версій JUnit, JUnit 5 складається з декількох різних модулів з трьох різних підпроектів.

JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*

Junit Platform платформа служить основою для запуску механізмів тестування на JVM. Він також визначає TestEngine API для розробки тестової структури, що працює на платформі. Крім того, платформа надає консольний запуск, щоб запустити платформу з командного рядка та створювати плагіни для Gradle і Maven , а також на базі JUnit 4 Runner для роботи TestEngine на платформі.

JUnit Jupiter - це комбінація нової моделі програмування та моделі розширення для написання тестів і розширень у JUnit 5. Підпроект Jupiter передбачає TestEngine тестування на платформі на базі Jupiter.

JUnit Vintage забезпечує TestEngine тестування JUnit 3 та JUnit 4 на платформі.

JUnit 5 вимагає Java 8 (або вище) під час виконання. Тим не менш, ви все ще можете перевірити код, який був скомпільований з попередніми версіями JDK.

Платформа JUnit (JUnit Platform) має наступні ідентифікатори артефактів:

- junit-platform-commons - внутрішня спільна бібліотека / утиліти JUnit. Ці утиліти призначені виключно для використання в рамках самого JUnit. Проте будь-яке використання зовнішніх сторін не підтримується.
- junit-platform-console - підтримка відкриття та виконання тестів на платформі JUnit з консолі.
- junit-platform-console-standalone - виконуваний JAR файл з усіма включеними залежностями надається у Maven Central у каталозі junit-platform-console-standalone.
- junit-platform-engine - публічний API для тестових движків(TestEngine).
- junit-platform-launcher - Загальнодоступний API для налаштування та запуску планів тестування - зазвичай використовуються IDE та інструментами для створення.
- junit-platform-runner - Runner для виконання тестів і тестових наборів на платформі JUnit у середовищі JUnit 4.
- junit-platform-suite-api - анотації для налаштування тестових наборів на платформі JUnit. Підтримується Engine'ом JUnitPlatform і сторонніми TestEngine реалізаціями.

- junit-platform-surefire-provider - підтримка відкриття та виконання тестів на платформі JUnit за допомогою Maven Surefire А такі артефакти має в собі JUnit Jupiter:
- junit-jupiter-api - API JUnit Jupiter для написання тестів та розширень .
- junit-jupiter-engine - JUnit Jupiter - реалізація тестового движка TestEngine, потрібна лише під час виконання.
- junit-jupiter-params - підтримка параметризованих тестів у JUnit Jupiter.
- junit-jupiter-migration-support - підтримка міграції від JUnit 4 до JUnit Jupiter потрібна лише для роботи вибраних правил JUnit 4.

JUnit Vintage має єдиний ідентифікатор - junit-vintage-engine - реалізація тестового двигуна JUnit Vintage, яка дозволяє запускати тести Junit, тобто тести, написані у стилі JUnit 3 або JUnit 4, на новій платформі JUnit.

JUnit Jupiter підтримує наступні анотації для налаштування тестів та розширення основ. Всі основні анотації розташовуються в org.junit.jupiter.api пакеті в junit-jupiter-api модулі.

Таблиця 3.1 Список анотацій JUnit 5

Анотація	**Опис**
@Test	Позначає той метод, що є методом для перевірки. На відміну від @Test анотації JUnit 4 , ця примітка не оголошує жодних атрибутів, оскільки тестові розширення в JUnit Jupiter працюють на основі власних анотацій. Такі методи успадковуються, якщо вони не перевизначені.
@ParameterizedTest	Позначає, що метод є параметризованим тестом . Такі методи успадковуються, якщо вони не перевизначені.
@RepeatedTest	Позначає, що метод є тестовим шаблоном для повторного тесту. Такі методи успадковуються, якщо вони не перевизначені.

Таблиця 3.1 (Продовження). Список анотацій JUnit 5

@TestFactory	Позначає, що метод є тестовою “фабрикою” для динамічних тестів. Такі методи успадковуються, якщо вони не перевизначені.
@TestInstance	Використовується для налаштування життєвого циклу тестового інстансу для анотованого тестового класу. Такі анотації успадковуються.
@TestTemplate	Позначає, що метод являє собою шаблон для тестових випадків, призначених для виклику кілька разів залежно від кількості контекстів викликів, що повертаються зареєстрованими провайдерами. Такі методи успадковуються, якщо вони не перевизначені.
@DisplayName	Оголошує відображуване ім'я для класу тесту або методу перевірки. Такі анотації не успадковуються.
@BeforeEach	Означає , що анотований метод повинен бути виконаний перед кожним @Test , @RepeatedTest, @ParameterizedTest, або @TestFactory методом, в поточному класі; аналогічний до анотації у JUnit 4 - @Before. Такі методи успадковуються, якщо вони не перевизначені.
@AfterEach	Означає, що анотований метод повинен бути виконаний після кожного @Test , @RepeatedTest, @ParameterizedTest, або @TestFactory метод , в поточному класі; аналогічний до JUnit 4 @After. Такі методи успадковуються, якщо вони не перевизначені.
@BeforeAll	Означає, що анотований метод повинен бути виконаний перед усіма @Test , @RepeatedTest, @ParameterizedTest і @TestFactory методами в поточному класі; аналогічно до JUnit 4 @BeforeClass. Такі способи успадковуються (якщо вони не приховані або не змінені), і вони повинні бути static (за винятком використання життєвого циклу тесту "per-class").

@AfterAll	Означає, що анотований метод повинен бути виконаний після всіх інших, що мають @Test , @RepeatedTest, @ParameterizedTesti @TestFactory методи в поточному класі; аналогічно до JUnit 4 @AfterClass. Такі способи успадковуються (якщо вони не приховані або не змінені), і вони повинні бути static (за винятком використання життєвого циклу тесту "на клас").
@Nested	Позначає, що клас анотованих є вкладеним, нестатичним тестовим класом. @BeforeAll і @AfterAll методи не можуть бути використані безпосередньо в @Nested класі тесту. Такі анотації не успадковуються.
@Tag	Використовується для оголошення тегів для тестування фільтрів на рівні класу або методу; аналогічні тестовим групам в TestNG або категорії в JUnit 4. Такі анотації успадковуються на рівні класу, але не на рівні методу.
@Disabled	Використовується для вимкнення класу тестів або методу тестування; аналогічно в JUnit 4 є @Ignore. Такі анотації не успадковуються .
@ExtendWith	Використовується для реєстрації власних розширень. Такі анотації успадковуються.

Методи анотовані @Test, @TestTemplate, @RepeatedTest, @BeforeAll, @AfterAll, @BeforeEach, @AfterEach анотаціями не повинні повертати значення.

Навіть якщо твердження можливості , що надаються JUnit Юпітеру досить для багатьох сценаріїв тестування, бувають випадки , коли більше потужності і додаткові функціональні можливості, такі як matchers бажані або необхідні. У таких випадках команда JUnit рекомендує використовувати сторонні бібліотеки затвердження, такі як AssertJ , Hamcrest , Truth та ін. Розробники можуть вільно використовувати бібліотеку затвердження за їх вибором.

Наприклад, комбінацію збірок і вільного API можна використовувати для того, щоб зробити твердження більш описовими та зрозумілими. Тим не менше, org.junit.jupiter.api.Assertions клас JUnit Jupiter не надає такого assertThat() методу, як у org.junit.Assert класі JUnit 4, який приймає Hamcrest Matcher. Замість цього розробникам пропонується використовувати вбудовану підтримку збірок, що постачаються сторонніми бібліотеками затвердження.

Наступний приклад демонструє, як використовувати assertThat() підтримку Hamcrest в тесті JUnit Jupiter. Поки бібліотека Hamcrest була додана в шлях до класів, ви можете статично імпортувати такі методи, як assertThat(), is(), і , equalTo(), а потім використовувати їх в тестах.

```
class HamcrestAssertionDemo {
  @Test
  void assertWithHamcrestMatcher() {
     assertThat(2 + 1, is(equalTo(3)));
  }
}
```

3.4 Інструменти для збірки проекту

Системи на базі Java можна збирати двома популярними системами: Maven та Gradle. Maven є найпоширенішим інструментом і має багато можливостей, але він для конфігурації проектів застосовує XML, що робить всі конфігурацію досить

великими і складними для читання. Окрім того, для створення більш складних алгоритмів по зборці проектів потрібно писати дуже багато конфігурацій та плагінів. Саме тому для збірки проекту був використаний Gradle — інструмент для збірки з наголосом на автоматизацію збірки та підтримку розробки на різних мовах програмування [16]. Gradle пропонує гнучку модель, яка може підтримувати весь життєвий цикл розробки від збору і упаковки коду до публікації веб-сайтів. Gradle був розроблений для забезпечення автоматизації збирання на декількох мовах і платформах, включаючи Java, Scala, Android, C / C ++ та Groovy, і тісно інтегрований з інструментами розробки і безперервних серверів інтеграції, включаючи Eclipse, IntelliJ і Jenkins.

Далі приведено список основник можливостей Gradle:

— В основі Gradle лежить багата розширювана мова предметної області (DSL), заснована на Groovy. Gradle ставить декларативну побудову проектів на наступний рівень, надаючи декларативні елементи мови, які можна застосовувати за власним вподобанням. Ці елементи також забезпечують підтримку збірки за конвенціями для Java [10], Groovy, Web і Scala проектів. Більш того, ця декларативна мова є розширюваною. Можна додавати власні елементи мови або покращувати вже існуючі, забезпечуючи короткі і зрозумілі збірки які просто підтримувати.

— Гнучкість і багатство Gradle дозволяють застосовувати загальні принципи проектування проектів. Наприклад, дуже легко скласти свою збірку з повторно використовуваних частин логіки для побудови інших проектів. З Gradle можна об’єднувати і розділяти компоненти за власним бажанням без необхідності підстраюватись до інструменту збірки.

— Підтримка Gradle для збірки кількох проектів краща серед усіх альтернатив. Залежності проекту перш за все. Завдяки Gradle є можливість налаштувати залежності як і в реальному світі, він слідує ієрархії проекту а не навпаки.

— Незважаючи на іноваційні підходи, Gradle дозволяє з легкістю мігрувати з інших систем збірки проектів таких як Maven та Ant.

— Скрипти збірки Gradle написані на Groovy, а не XML. Це дозволяє використовувати повноцінну динамічну мову для опису своїх збірок, що дає незрівнянну гнучкість та лаконічність усіх скриптів.

— Для використання Gradle не обов'язково встановлювати його собі. Gradle надає можливість використовувати спеціально побудовану обгортку, що зменшує кількість необхідних залежностей і дозволяє простіше виконувати збірки на сервісах для Continuous Integration (CI).

Висновки до розділу 3

Отже, для розробки пропонованого рішення були використані перевірені часом технології та фреймворки, що мають широку підтримку від розробників та надійну інтеграцію з іншими інструментами. А саме – IntellijIDEA, Maven, Gradle, а також бібліотека JUnit 5, яка й була розширена.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Структура програмного рішення

Пропоноване програмне рішення являє собою розширення бібліотеки JUnit 5. Не дивлячись на те, що в Java є багато інноваційних функцій, однією із найцікавіших і уже існуючих вбудованих функцій є багатопоточне програмування. Багатопоточна програма містить 2 або більше частини, що можуть виконуватися одночасно. Кожна така частина називається потоком, а кожен поток визначає окремий шлях виконання. Таким чином, багатопоточність є спеціалізованою формою багатозадачності[6]. Ця можливість Java і ввійде в основу пропонованого рішення прискорення модульного тестування.

За замовчуванням тести JUnit Jupiter запускаються послідовно в одному потоці (базовий двигун (Engine), за замовчуванням). Запуск тестів паралельно, наприклад, для прискорення виконання, доступний як функція на вибір. Але, по перше, ця функція - експериментальна, а по-друге, вона доступна лише після версії 5.3.

Після ввімкнення engine JUnit Jupiter виконає тести на всіх рівнях тестового дерева повністю паралельно відповідно до наданої конфігурації під час спостереження за декларативними механізмами синхронізації. Проте, як це відомо, для деяких програм паралельні тести не мають сенсу у певних частинах. Адже ми не можемо одночасно тестувати створення, модифікацію і видалення даних - оскільки ми можемо отримати не консистентні дані. Наприклад, якщо в наших тестах буде сценарій для додавання нового користувача та видалення користувача, то дані для видалення ще не будуть створені в разі паралельного виконання тестів.

Пропоноване ж рішення дає змогу, по-перше, запускати не всі, а обрані тести паралельно; по-друге, користуватися ним для більш ранніх версій JUnit 5.

Його використання допомагає зменшити час на тестування та ефективно використати ресурси.

Окремо був розширений engine, який запускає тести позначені пропонованою анотацією. При його розробці використовується модель "fork-join": при паралельних обчисленнях модель "fork-join" є способом налаштування та виконання паралельних програм, так, що виконання відбувається паралельно в певних точках програми, "приєднується" (merge) в наступній точці та відбувається послідовне виконання. Паралельні розділи можуть ділитися рекурсивно, доки не досягнуть певної деталізації (granularity)[9]. Її метою є використання всієї доступної потужності обробки для підвищення продуктивності програми.

Отже, по-перше, я додала в JUnit 5 власну анотацію @AsyncTest для маркування модульних тестів, які потрібно запустити паралельно.

Далі я створила імплементацію інтерфейса TestEngine - клас AsyncTestEngine, що наслідується від абстрактного класу HierarchicalTestEngine.

Інтерфейс LauncherDiscoveryRequest сканує класи та методи, що зрозуміти, що являється тестом - збирає результат у чергу.

Тобто Launcher перед стартом тестів шукає реалізації інтерфейсу TestEngine. Робить він це за допомогою classLoader, що завантажує клас в JVM.

Отримавши результати сканування, Launher передає його в HierarchicalTestEngine (через ExecutorRequest, де тести- вхідні дані) - там починається виконання executorService(), також доданого мною для AsyncTestEngine.

Таким чином, результати фільтруються відповідно до анотації, якою позначений тест.

Тести зі стандартною анотацією @Test йдуть на виконання та виконуються послідовно через JupiterTestEngine.

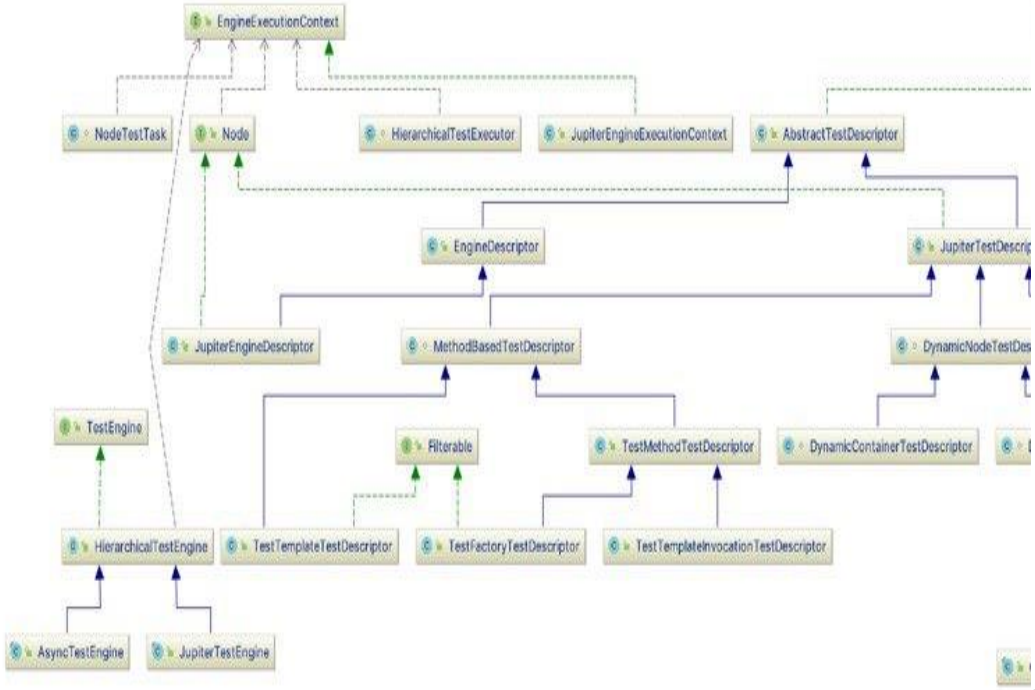
А от тести , позначені новою анотацією @AsyncTest - через AsyncTestEngine. ForkJoinPool, застосований у AsyncTestEngine розпаралелює тести, які були позначені відповідною анотацією.

Завдяки цьому методи необхідні тести можуть маркуватися для одночасного запуску, що значно зекономить ресурси та підвищить використання всієї доступної швидкодії обробки, покращивши продуктивність програми.

4.2 Діаграма класів

В діаграмі класів (рисунок 4.1 і 4.2) видно, що AsyncTestEngine клас імплементує TestEngine

INCLUDEPICTURE "https://lh6.googleusercontent.com/dWwPeN2cxxT5KOXYKecDgaXfrL8obg--iREVshhKO_ZrKKyY4ohbF1dJMrnsPUyKjKhgHjOJ-jUGQsI9vB0pgSznjd231-omXz_mPQEh5ecTxgWCO1lUNvBepSsYQwdaZpzVJAOQ" * MERGEFORMATINET



->

Рисунок 4.1. Діаграма класів JUnit 5 з розширенням Async

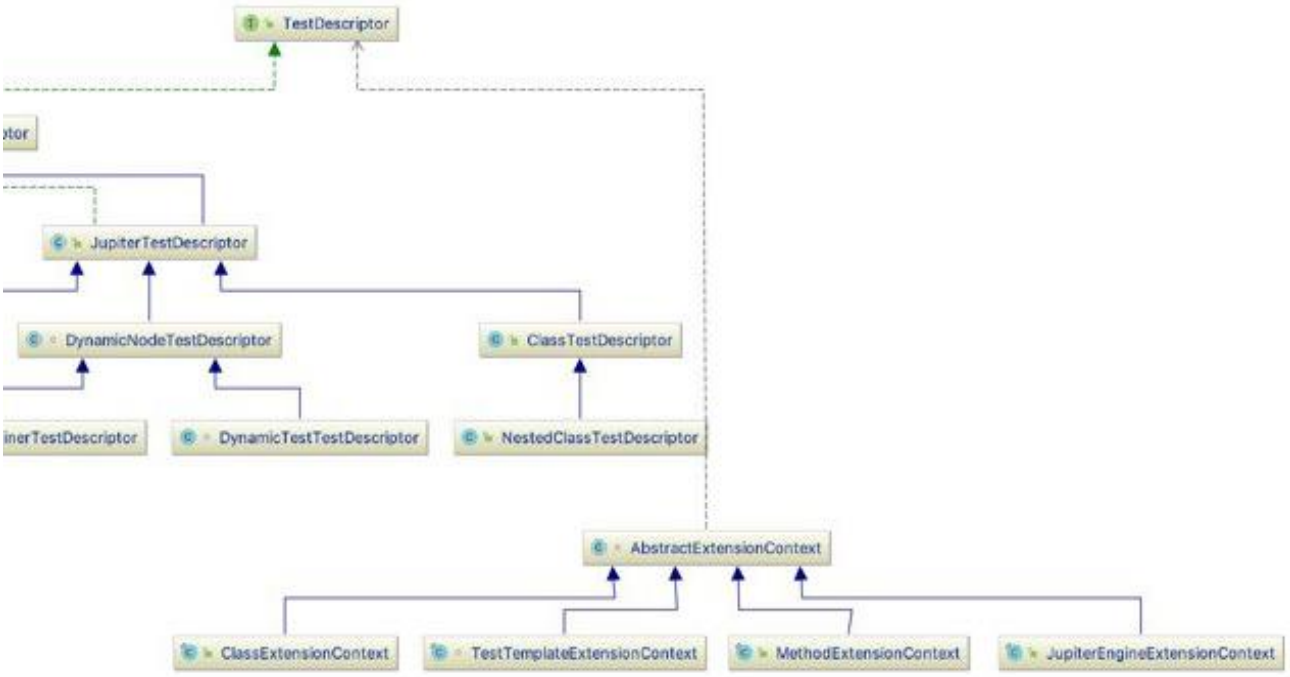


Рисунок 4.2. Діаграма класів JUnit 5 (інша частина)

Висновки до розділу 4

Таким чином рішення імплементує та розширяє можливості існуючої бібліотеки , за допомогою створеної анотації можна помічати код, який потрібно тестувати у паралельному режимі

5 МЕТОДОЛОГІЯ РОБОТИ ТА РЕЗУЛЬТАТИ ОБЧИСЛЮВАЛЬНИХ ЕКСПЕРИМЕНТІВ

Для дослідження було взято проект, в якому міститься до 1500 модульних тестів. Це модульні тести, що написані для тестування функціоналу Java Core, наприклад, масивів.

До цього проекту було підключено бібліотеку JUnit розширену Async.

Далі одні й ті ж тести позначились анотацією @Test та окремо анотацією @AsyncTest.

Те ж сааме мають зробити розробники – підключити бібліотеку та

Перші були виконані послідовно. А другі – паралельно.

Результати видно на знімках екрану нижче (рисунок 5.1, 5.2).

Тож було запущено 611 тестів спочатку послідовно:

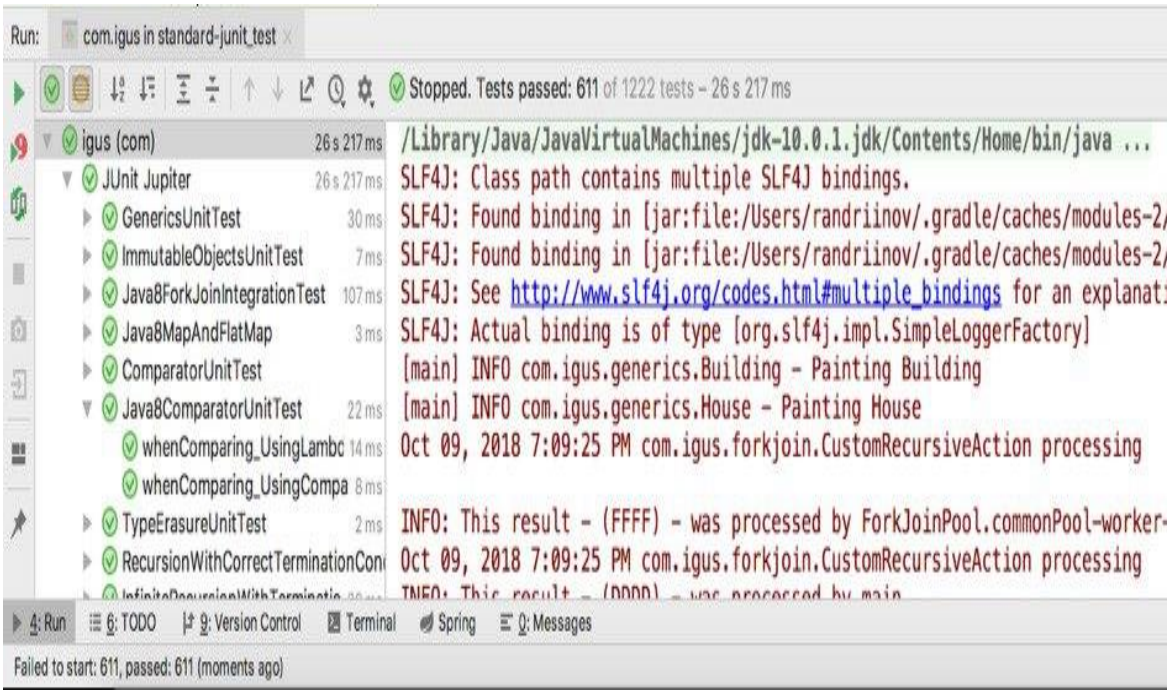


Рисунок 5.1. Результат виконання послідовних тестів

а потім паралельно:

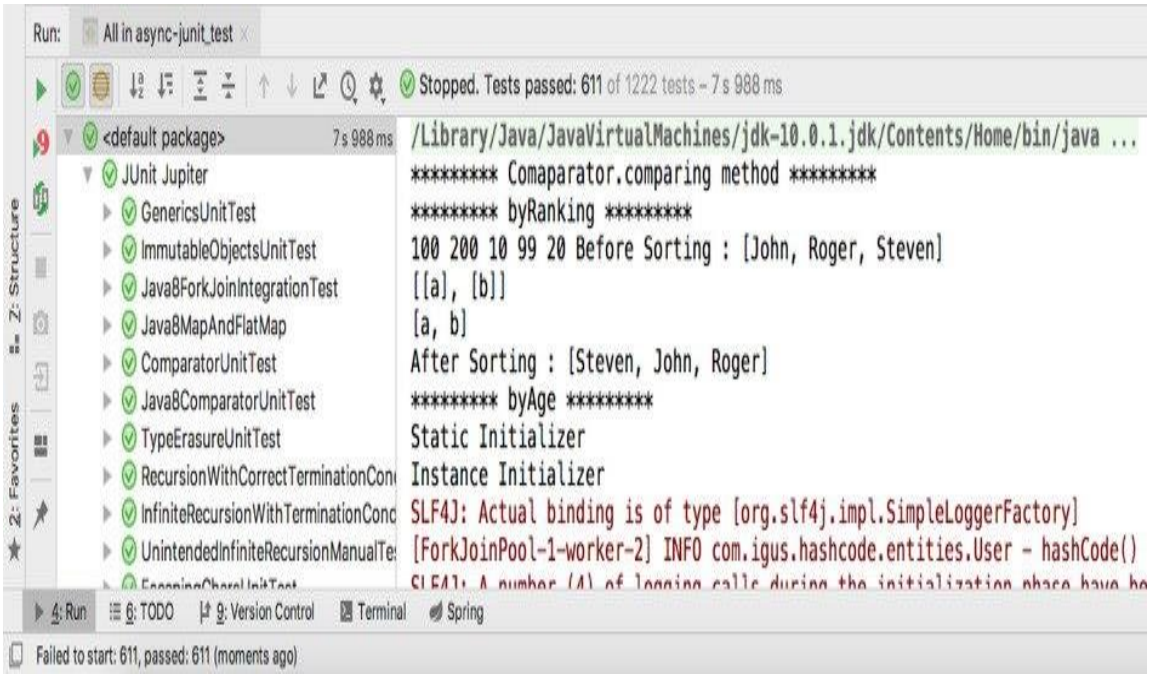


Рисунок 5.2. Результат виконання паралельних тестів

В результаті, видно, що при одночасному запуску швидкість виконання 611-ти тестів складає 7 сек 988 мсек, що швидше, аніж 26 сек 217 мсек при послідовному запуску.

Висновки до розділу 5

Отже, було створено анотацію та реалізовано новий клас AsyncEngine який імплементує TestEngine інтерфейс (базовий у JUnit 5) з використанням моделі “fork-join”. Дослідження показали, що використання такого поєднання значно прискорило виконання тестів.

6 РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

В останні роки в Україні і в світі значно розивнувся ринок програмних продуктів. Через це розробники мають працювати в умовах конкуренції. Тому в такому середовищі, ще до пропозиції певного програмного забезпечення, важливим є аналіз існуючих рішень, порівняння з пропонованим та визначення його конкурентоспроможності, техніко-економічне обгрунтування та ефективнісь пропонованого продукту.

У рамках цих завдань було проаналізовано ринок існуючих рішень, зроблено аналіз ринкових можливостей запуску стартап-проекту.

Кожен замовник програмного продукту сплачує кошти, наймає спеціалістів, сподіваючись отримати готовий продукт у найвищій якості, оскільки це може напряму вплинути на його справу, його прибутки та можливості зростання. В свою чергу, команда, що займається розробкою має це розуміти і робити все можливе для задоволення вимог замовника.

6.1. Опис ідеї стартап-проекту

Крім якості, напряму на бізнес замовника впливає час виконання роботи, адже від того, скільки часу потребуватиме розробник на доведення продукту чи компоненту до стадії готовності, залежатиме розмір оплати за ресурси. І якщо, завдяки певним рішенням, ресурси можна використовувати раціональніше — це дасть змогу зменшити витрати.

Відомо, що сьогодні на більшості проектів впроваджено практику модульного тестування, яка заздалегідь, на ранніх етапах виявляє помилки та дефекти — навіть до завершення написання кода. Це також економить ресурси, оскільки не витрачається час на всі етапи (розробка, code review, тестування) та час роботи інших спеціалістів, які можуть ідентифікувати цю помилку. Крім того,

завдяки модульному тестуванню відсікається більшість помилок, щоб вони не опинилися в уже готовому продукті. Для цього використовуються бібліотеки модульного тестування.

Так, наприклад, для мови Java найпопулярнішою бібліотекою є JUnit. Незважаючи на попит та багато корисних функцій — JUnit неідеальний, а недолік в тому, що він не пропонує вбудоване рішення, яке б змогло запустити модульні тести у багатопоточному режимі.

Рішення, що пропонується, допоможе розширити можливості найвідомішої бібліотеки для юніт-тестування на Java. Завдяки розширенню, розробник зможе позначити необхідні тести анотацією, яка вкаже на паралельний запуск цих тестів. Завдяки одночасному запуску отримаємо економію ресурсів.

Таблиця 6.1. Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
зменшення часу на виконання модульних тестів завдяки паралельному їх виконанню у JUnit 5 за допомогою створення кастомної java-анотації (з можливістю відмітити всі або деякі тести кастомною анотацією) та розширенням функціоналу бібліотеки	1. модульне тестування програмного забезпечення	2. Раціональне використання ресурсів та коштів
	3. збірка великих проектів	1. Значне зменшення часу на виконання юніт-тестів та відповідно прискорення збірок

Оскільки послідовний запуск тестів не завжди є раціональним у використанні часових ресурсів, то з*апропоноване* програмне рішення щодо процесів модульного тестування, *яке полягає* у зменшенні часу на виконання модульних тестів завдяки

паралельному їх виконанню у JUnit 5 за допомогою створення кастомної java-анотації (з можливістю відмітити всі або деякі тести кастомною анотацією) та розширенням функціоналу бібліотеки, який відсканує і запустить всі тести, позначені анотацією в багатопоточному режимі (одночасно).

Проводиться порівняльний аналіз показників: для власної ідеї визначаються показники, що мають:

а) гірші значення (W, слабкі);

б) аналогічні (N, нейтральні) значення;

в) кращі значення (S, сильні) (таблиця 6.2) .

Таблиця 6.2. Визначення сильних, слабких та нейтральних характеристик

№	Техніко-економічні характеристики ідеї	Продукція конкурентів		W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	JUnit 5			
1.	Паралельний запуск модульних тестів	+	-	-	-	+
2.	Вибіркове позначення анотаціями	+	-	-	-	+
3.	Розширюва-ність	+	+	-	+	-
4.	Відстеження прогресу	+	-	-	+	-

6.2 Технологічний аудит ідеї проекту

В межах даного підрозділу необхідно провести аудит технології, за допомогою якої можна реалізувати ідею проекту. Визначення технологічної здійсненності ідеї проекту передбачає аналіз таких складових (таблиця 6.3):

— за якою технологією буде виготовлено товар згідно ідеї проекту;

— чи існують такі технології, чи їх потрібно розробити/доробити;

— чи доступні такі технології авторам проекту;

— чи є фінансово затратним використання таких технологій;

— чи можливо замінити дані технології іншими, менш затратними;

— як вплине зміна технологій на функціональність продукту та його якість.

Таблиця 6.3. Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Паралельний запуск модульних тестів	Мова Java	Наявна	Технологія з відкритим кодом
2	Використання бібліотеки	JUnit 5	Наявна	Бібліотека з відкритим кодом
3	Використання розробниками та QA-інженерами	Середовище розробки Intellij IDEA	Наявна	Технологія з відкритим кодом
4	Вибіркове зчитування анотацій	Reflection	Наявна	Відкритий алгоритм
Висновок: проект реалізувати можливо. Обрана технологія реалізації ідеї проекту: мова програмування Java, бібліотека для розширення JUnit 5				

За результатами аналізу таблиці робиться висновок щодо можливості технологічної реалізації проекту: так чи ні, а також технологічного шляху, яким це доцільно зробити (з поміж названих технологій обираються такі, що доступні авторам проекту та є наявними на ринку).

6.3 Аналіз ринкових можливостей запуску стартап-проекту

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів.

Спочатку проводиться аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (таблиця 6.4).

Таблиця 6.4. Попередня характеристика потенційного ринку стартап-проекту

№	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	7
2	Загальний обсяг продаж, грн/ум.од	900 грн
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі (або по ринку), %	45 %

Середня норма рентабельності в галузі (або по ринку) порівнюється із банківським відсотком на вкладення. За умови, що останній є вищим, можливо, має сенс вкласти кошти в інший проект.

За результатами аналізу таблиці робиться висновок щодо того, чи є ринок привабливим для входження за попереднім оцінюванням. Цільовим ринком може бути навчальні заклади, приватні установи та окремі особи. Користувач може з

легкістю вивчити нові слова завдяки декільком видам сприйняття, такі як візуальне (асоціація слова з зображенням), аудиальне (можливість послухати вимову слова), моторне (написання слова та перекладу) та вимовне (тренування правильній вимові іноземних слів).

Надалі визначаються потенційні групи клієнтів, їх характеристики, та формується орієнтовний перелік вимог до товару для кожної групи.

Дані щодо всіх можливих клієнтів, їх характеристик наведені у таблиці 6.5.

Таблиця 6.5. Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Якість програмного продукту, що розробляється	Споживачі — організації або особи, яким необхідне проведення модульного тестування	Компанії заключають довготривалі договори, а стартапери віддають перевагу пробному терміну	Можливість запуску модульних тестів паралельно; Можливість тестування з різними версіями бібліотеки; Можливість вибіркового запуску тестів; Зручність у використанні.

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (таблиці 6.6-6.7).

Надалі проводиться аналіз пропозиції: визначаються загальні риси конкуренції на ринку. Аналіз пропозиції необхідно виконати аналізуючи існуючі види конкуренцій.

Таблиця 6.6. Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії

1	Інтеграція під нові проекти	Потребує інтеграції зі сторони збірника	Залучення спеціаліста предметної області; Повернення до попередніх версій
2	Вразливість до несанкційованого втручання	Потребує надбудови захисту інформації	Залучення команди спеціалістів із захисту інформації Залучення команди веб-розробників
3	Домоміжний функціонал	Сервіс має визначений функціонал	Залучення команди IT- спеціалістів для додавання нового функціоналу

Таблиця 6.7 описує фактори можливостей системи, тобто наскільки розроблене програмне забезпечення є гнучким у використанні.

Чи можна реалізувати можливість системи до самонавчання. Ведеться опис можливої реакції компанії на ідею розробки автоматизовану систему під різні платформи.

Таблиця 6.7. Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Незалежність від версії бібліотеки	Можна використовувати різні версії бібліотеки та інтегрувати зі сторонніми бібліотеками	Вихід на різні ринки IT
2	Незалежність від версії мови програмування Java	Працює з різними версіями – не потрібно міняти проект, щоб використати рішення	Модифікація існуючих систем

Аналіз пропозицій зображено на таблиці 6.8.

Таблиця 6.8. Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - монополія/олігополія/ монополістична/чиста	Чиста	Презентація продукту на виставках, конференціях
2. За рівнем конкурентної боротьби - локальний/національний/...	національний	Рекламування веб-сервісу міжнародних сайтах
3. За галузевою ознакою - міжгалузева/ внутрішньогалузева	міжгалузева	Використовувати в різних галузях виробництва
4. Конкуренція за видами товарів - товарно-родова - товарно-видова - між бажаннями	товарно-видова	Розповідати про свої переваги перед конкурентом у цій галузі
5. За характером конкурентних переваг - цінова / нецінова	Нецінова	Надання функцій, які відсутні у конкурентів, модифікація функцій, що мають конкуренти
6. За інтенсивністю - марочна/не марочна	Марочна	Надання функцій, які відсутні у конкурентів, модифікація функцій, що мають конкуренти

Після аналізу конкуренції проводиться більш детальний аналіз умов конкуренції в галузі (таблиця 6.9).

На основі аналізу конкуренції, проведеного в п. 1.5 (таблиця 6.9), а також із урахуванням характеристик ідеї проекту (таблиця 6.2), вимог споживачів до товару (таблиця 6.5) та факторів маркетингового середовища (таблиця 6.6 - 6.7) визначається та обґрунтовується перелік факторів конкурентоспроможності. Аналіз оформлюється за таблицею 6.10.

Таблиця 6.9. Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	JUnit5	TestNG	Open Source	Java розробники	Лояльність клієнтів

Таблиця 6.10. Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Надана можливість паралельно запускати модульні тести	Існуючі конкуренти або не такої мають можливості, або реалізація цього функціоналу не є конкурентоспроможною.

За визначеними факторами конкурентоспроможності (таблиця 6.10) проводиться аналіз сильних та слабких сторін стартап-проекту (таблиця 6.11)

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (таблиця 6.12) на основі виділених ринкових загроз та можливостей, та сильних і слабких сторін (таблиця 6.11).

Таблиця 6.11. Порівняльний аналіз сильних та слабких сторін

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з Database Generator (даним продуктом)						
			-3	-2	-1	0	1	2	3
1	Надана можливість паралельно запускати модульні тести	20	+						

Для здійснення SWOT-аналізу на підприємстві необхідне відповідне інформаційне забезпечення, яке повинно включати: базу даних; методи та моделі, необхідні для SWOT-аналізу; набір організаційних і методичних прийомів, необхідних для підвищення надійності інформаційного забезпечення.

Методика SWOT-аналізу ґрунтується на підході, який дає змогу вивчати зовнішнє і внутрішнє середовище підприємства разом.

За допомогою цієї методики можна встановити взаємозв'язки між силою та слабкістю.

Таблиця 6.12. SWOT-аналіз стартап-проекту

Сильні сторони (S): Паралельний запуск модульний тестів Оперування великими данними Підтримка 3х сторін Розширюваність	Слабкі сторони (W): Залежність від основної бібліотеки
Можливості (O): Можливість підтримки сторонніх бібліотек Зручність у використанні	Загрози (T): Втрата даних

Перелік ринкових загроз та ринкових можливостей складається на основі аналізу факторів загроз та факторів можливостей маркетингового середовища. Ринкові загрози та ринкові можливості є наслідками (прогнозованими результатами) впливу факторів, і, на відміну від них, ще не є реалізованими на ринку та мають певну ймовірність здійснення.

Перелік ринкових загроз та ринкових можливостей складається на основі аналізу факторів загроз та факторів можливостей маркетингового середовища (рисунок 6.13). Ринкові загрози та ринкові можливості є наслідками (прогнозованими результатами) впливу факторів, і, на відміну від них, ще не є реалізованими на ринку та мають певну ймовірність здійснення.

Таблиця 6.13. Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Орієнтація поточної моделі на ринок приватних підприємств	30 %	35 год

2	Орієнтація поточної моделі на ринок державних установ	27 %	77 год
3	Орієнтація на приватних осіб	35 %	155 год
4	Орієнтація на розробку серверної частини	65 %	115 год
5	Орієнтація на веб-розробку	45 %	97 год

Альтернатива, де отримання ресурсів є більш простим та ймовірним — №4 "Переорієнтація на розробку серверної частини", що становить 65 відсотків. Це значення перевищує інші альтернативи.

Альтернатива, де строки реалізації є більш стислими — №2 “Орієнтація поточної моделі на ринок приватних підприємств”. Терміни реалізації в цьому разі становлять 35 годин.

6.4 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (таблиця 6.14).

Таблиця 6.14. Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Приватні підприємствава	Готові	Високий	Висока	Просто

2	Державні установи	Потребують недовгих переговорів	Середній	Середня	Складно
3	Стартапери	Потребують довгих переговорів	Низький	Низька	Дуже складно

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (таблиця 6.15). За результатами аналізу потенційних груп споживачів (сегментів) автори ідеї обирають цільові групи, для яких вони пропонуватимуть свій товар, та визначають стратегію охоплення ринку.

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів.

Таблиця 6.15. Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення Ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
Орієнтація поточної моделі на ринок приватних підприємств	Стратегія концентрованого маркетингу	Приватні підприємства потребують якості роботи, яку надає підтримка декількох платформ даним продуктом	Стратегія спеціалізації (спирається на диференціацію)

Перелік ринкових загроз та ринкових можливостей складається на основі аналізу факторів загроз та факторів можливостей маркетингового середовища. Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту.

Наступним кроком є вибір стратегії конкурентної поведінки (таблиця 6.16).

Таблиця 6.16. Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів	Чи буде компанія копіювати основні характеристики конкурента	Стратегія конкурентної поведінки
Ні	Так	Так	Стратегія заняття конкурентної ніші

6.5 Аналіз ринкових можливостей запуску стартап-проекту

Для цього у таблиці 6.17 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Після формування маркетингової моделі товару слід особливо відмітити — чим саме проект буде захищено від копіювання.

Захист може бути організовано за рахунок захисту ідеї товару (захист інтелектуальної власності), або ноу-хау, чи комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.

Для цього можна скористатись правом на оформлення патенту чи авторського права. Будь-яке ноу-хау, тобто те інноваційне, що розроблено, не потрібно розкривати у заявці на отримання патенту, але загальні особливості потрібно описати.

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар. Проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту). Також необхідно визначити ринок та інтереси користувачів відносно кожного сезону,

відповідно вікових категорій та різних регіонів країни. Проаналізувати ринок та цінову політику конкурентів, можливі витрати на підтримку рекламної кампанії для кращого попиту на послуги, шо надає програмний продукт.

Таблиця 6.17. Визначення ключових переваг концепції потенційного товару

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
Швидкість роботи та оптимізація алгоритму	Можливість перегляду прогресу навчання	Конкуренти або не мають орієнтованості на аналіз навчання, або здійснюють неповний аналіз

Надалі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (таблиця 6.18).

М/Нм — монотонні або немонотонні;

Вр/Тх/Тл/Е/Ор — вартісні, технічні, технологічні, ергономічні або органолептичні (останній — для продуктів харчування)

Після формування маркетингової моделі товару слід особливо відмітити — чим саме проект буде захищено від копіювання.

Захист може бути організовано за рахунок захисту ідеї товару (захист інтелектуальної власності), або ноу-хау, чи комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар (таблиця 6.19).

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (таблиця 6.20):

— проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту);

— вибір та обґрунтування оптимальної глибини каналу збуту;

— вибір та обґрунтування виду посередників;

— вибір та обгрунтування потреб користувачів в даному регіоні;

— вибір та обгрунтування цінової політики.

Таблиця 6.19. Визначення меж встановлення ціни

№	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	25…515 грн	100…700 грн	27000…52000 грн	33…108 грн

Таблиця 6.20. Формування системи збуту

№	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Система повинна надаватися в режимах пробної версії та повної сплатити після закінчення випробувального строку	Проста та легкість в експлуатації	Веб-сайт	Збут за допомогою посередника

Останньою складової маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів (таблиця 6.21).

Маркетингова програма, як правило, спрямована на вирішення окремих комплексних проблем, наприклад на організацію виробництва нового продукту, на завоювання нового сегмента або ринку в цілому. Слід мати на увазі, що специфіка

маркетингової стратегії зумовлює і специфіку програми. Розробляючи програму, необхідно насамперед враховувати ключові фактори комерційного успіху, чітко розрізняючи об'єктивні зовнішні обмеження. Перед початком нового проекту потрібно проаналізувати ринок даної області, свідомо прийняти рішення чи дійсно це необхідно.

Таблиця 6.21. Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Продаж програми через авторизовану мережу	Веб-сайти	Вивчення нових слів завдяки декільком видам сприйняття.	Довести, що програмний продукт оптимально аналізує процес навчання	Залучення рекламних компаній, робити упор на самонавчання

Висновки до розділу 6

Програмне забезпечення має переваги над існуючими конкурентами та є конкурентноздатним на ринку.

Для модульних тестів на Java найчастіше використовують бібліотеку JUnit. Близько року назад вийшла 5-та версія з багатьма перевагами, які спонукають

розробникив переходити на неї зі старіших версій. Проте за замовчуванням тут забезпечено виконання модульних тестів лише послідовно.

А отже, не вистачає функціоналу, який би прискорив час на виконання тестів, запустивши їх у паралельному режимі. Саме тому розроблений продукт є конкурентоспроможним та потенційно цінним для розробників.

ВИСНОВКИ

Отже, було описано модульне тестування та проаналізовано відомі інтрументи, бібліотеки, які використовуються в процесі.

В результаті дипломної роботи була побудовано програмне рішення, що розширило та покращило існуючу бібліотеку. Був створений окремий модуль та внесені відповідні зміни в існуючий проект.

Тестування програмного забезпечення має бути не лише корисним, але й максимально раціональним. На сьогоднішній день практично у кожному прооекті, що прагне максимальної якості, розробляються юніт-тести. Вони запускаються щоразу разом зі збіркою спочатку на машині розробника, що вносить зміни, щоб провалідувати код, перш ніж він потрапить далі. Потім знову йде перевірка - на сервері. Все це займає час.

Для модульних тестів на Java найчастіше використовують бібліотеку JUnit. Близько року назад вийшла 5-та версія з багатьма перевагами, які спонукають розробникив переходити на неї зі старіших версій. Проте за замовчуванням тут забезпечено виконання модульних тестів лише послідовно. А отже, не вистачає функціоналу, який би прискорив час на виконання тестів, запустивши їх у паралельному режимі.

Беззаперечним плюсом багатопоточності є можливість запускати потоки у паралельному режимі, скорочуючи тим самим час на їх виконання. Це можна використати, у тому числі, і у застосуванні для реалізації засобів для модульного тестування.

Також я застосувала fork join - метод, що слугує для збільшення продуктивності виконання великої кількості робочих завдань і полягає в тому, що кожна задача розбивається на безліч дрібніших синхронізованих завдань, які обробляються паралельно на різних серверах. І оскільки метою є скорочення часу на виконання – то саме цей метод став корисним.

Крім того, у моїй роботі був використаний механізм рефлексії, адже використання анотацій стало однією із ключових особливостей пропонованого рішення. А зчитування анотацій без цього механізму практично неможливе.

Для розробки пропонованого рішення були використані перевірені часом технології та фреймворки, що мають широку підтримку від розробників та надійну інтеграцію з іншими інструментами. А саме – IntellijIDEA, Maven, Gradle, а також бібліотека JUnit 5, яка й була розширена.

Отже, дослідження показали, що використання такого поєднання значно прискорило виконання тестів.

За результатами виконання тестів підтверджена коректність отриманих результатів, отже система відповідає поставленим вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Foundations of Software Testing: ISTQB Certification / [Dorothy Graham, Erik Van Veenendaal, Isabel Evans].//Cengage Learning EMEA.- 2008
2. John F. Dooley Galesburg, Software Development, Design and Coding Illinois, USA.- 2017
3. Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
4. Test Doubles - [Електронний ресурс] — Режим доступу: https://uk.wikipedia.org/wiki/Test_double#Spy
5. Мокито [Електронний ресурс] Режим доступу: https://dzone.com/refcardz/mockito?chapter=1
6. EasyMock Basics [Електронний ресурс] Режим доступу: https://www.tutorialspoint.com/easymock/easymock_overview.htm
7. IntelliJ IDEA the Java IDE [Електронний ресурс] — Режим доступу: https://www.jetbrains.com/idea/
8. Бодягин І. Епоха паралельності. Способи виживання в епоху многоядерного паралелізму // RSDN Magazine. - The RSDN Group, 2009. - № 3 .
9. Structured Parallel Programming: Patterns for Efficient Computation/ [Michael McCool; James Reinders; Arch Robison].// Elsevier. - 2013.
10. Блох Д. Java. Эффективное программирование / Джошуа Блох — М. Лори, 2014. — 310 с.
11. Сиерра К., Бейтс Б. Изучаем Java / Кэти Сиерра, Берт Бейтс — М: Эксмо, 2016. — 720 с.
12. Хорстманн К. Java SE8. Вводный курс / Кей Хорстманн. — М: Вильямс, 2014. — 208 с.
13. Шефер К., Хо К., Харроп Р. Spring 4 для профессионалов / Крис Шефер, Кларенс Хо, Роб Харроп — М. Вильямс, 2016. — 752 с.

14. Эккель Б. Философия Java. Полное издание: Java / Брюс Эккель. — СПб: Питер, 2003. — 976 с.
15. Amazon Web Services [Електронний ресурс] — Режим доступу: https://aws.amazon.com
16. Gradle Build Tool [Електронний ресурс] — Режим доступу: https://gradle.org
17. Sujoy Acharya - Mockito for Spring – Packt Publishing 2017
18. Т.С. Ігушкіна, Д.С. Смаковський - “Програмні засоби прискорення модульного тестування” – СТАЛИЙ РОЗВИТОК — XXI СТОЛІТТЯ:УПРАВЛІННЯ, ТЕХНОЛОГІЇ, МОДЕЛІ - Дискусії 2018 Колективна монографія с.416-421
19. Н.Блінов, В.СРоманчик, Java Методи Програмування Видавництво «Четыре Четверти», 2013
20. Стив Макконнелл, Совершенный код - Microsoft Press, 2017. – 896 с.
21. Джошуа Кериевски, Рефакторинг с использованием шаблонов М.: Издательский дом «Вильямс», 2016. — 400 с.
22. Мартин Фаулер Рефакторинг. Улучшение существующего кода – Символ-Плюс, 2008. – 432 с.
23. Роберт К. Мартин Быстрая разработка программ. Принципы, примеры, практика — М.: Издательский дом «Вильямс», 2004. — 752 с.
24. Мартин Фаулер Архитектура корпоративных программных приложений М.: Издательский дом «Вильямс», 2007. — 544 с.
25. Bruce Eckel, Thinking in Java 4th edition, Prentice Hall 2006
26. Prabath Siriwardena Mastering Apache Maven 3 Packt Publishing, pp. 298. 2014
27. Brett Spell, Pro Java 8 Programming, 3rd Edition, Apress 695p, 2015
28. Uttam Kumar Roy, Advanced Java Programming, 880 p.,2015
29. Alan Mycroft, Mario Fusco, Raoul-Gabriel Urma, Java 8 in Action, Meaning 2014
30. Test-Driven Java Development By Viktor Farcic and Alex Garcia – Packt Publishing, pp. 98. 2015

31. Core Java : Complete Reference for the Really Impatient. By Harry. H. Chaudhary.- Harry & Associates -2014
32. Mastering Unit Testing Using Mockito and JUnit By Sujoy Acharya – PacktPublishing. – 2014
33. J.Paniza - Rapid Java Web Development By Javier Paniza - PacktPublishing. – 2011
34. Java™ Programming: A Complete Project Lifecycle Guide By Nitin Shreyakar - Lulu.com.- 2015
35. Johannes Link and Peter Fröhlich - Unit Testing in Java: How Tests Drive the Code //. - Elsevier Science. – 2015
36. Tilo Linz - Testing in Scrum: A Guide for Software Quality Assurance in the Agile World - Rocky Nook RELEASED: Mar 28, 2014
37. Gerardus Blokdyk - Unit testing A Clear and Concise Reference - 5STARCooks: Apr 5, 2018
38. Gerardus Blokdyk - Quality Assurance Complete Self-Assessment Guide - The Art of Service RELEASED: Jan 5, 2018
39. B. Walraet - Programming, The Impossible Challenge - Elsevier Science: Jun 28, 2014
40. Viktor Farcic and Alex Garcia - Test-Driven Java Development - Packt Publishing: 2015
41. Harry. H. Chaudhary - Java : The Complete Reference.- Harry & Associates. R-: Aug 15, 2014
42. Gerardus Blokdyk - Java Reflection Complete Self-Assessment Guide. - The Art of Service: Jan 5, 2018

ДОДАТОК А

Застосування мікросервісної архітектури для побудови відмовостійкої хмарної систем

Апробації

УКР.НТУУ “КПІ”.ТВ-з7127мп

Аркушів

2018

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Національний університет “Києво-Могилянська академія”
Вища економіко-гуманітарна школа (Польща)

СТАЛИЙ РОЗВИТОК — XXI СТОЛІТТЯ: УПРАВЛІННЯ, ТЕХНОЛОГІЇ, МОДЕЛІ

Дискусії 2018

Колективна монографія

Київ, Україна
2018

4.11. Програмні засоби прискорення модульного тестування[67]

Вступ. Кожен замовник програмного продукту сплачує кошти, наймає спеціалістів, сподіваючись отримати готовий продукт найвищої якості, оскільки це може напряму вплинути на його справу. В свою чергу, команда, що займається розробкою, має це розуміти і робити все можливе для задоволення вимог замовника.

У процесах розробки програм існують різні підходи, проте кожен націлений на досягнення максимально якісного кінцевого продукту. В цьому розробникам допомагають різного роду перевірки та аналіз коду. Тест — це процедура, яка дає можливість підтвердити або спростувати роботоздатність програмного коду. В ньому містяться перевірки умов, які можуть або виконуватися або не виконуватися. У першому випадку тест вважається пройденим (passed), а в другому — ні (failed). Пройдений тест підтверджує саме ту поведінку кода, яку передбачав розробник. Саме тому тести та їхні результати є важливим показником якості готового програмного продукту.

Постановка проблеми й актуальність. Сучасні проекти важко уявити без тестування на різних рівнях: так, існує модульне тестування (unit testing), інтеграційне (integration testing), системне (system testing) та приймальне тестування (UAT testing)[68] (рис. 1).

67 **Автори Ігушкіна Т.С., Смаковський Д.С.**
68 Foundations of Software Testing: ISTQB Certification / [Dorothy Graham, Erik Van Veenendaal, Isabel Evans].//Cengage Learning EMEA. — 2008

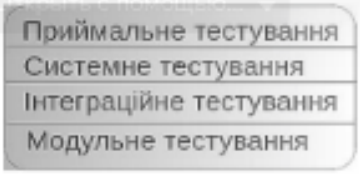


Рисунок 1. Схематичне зображення рівнів тестування

Модульні тести — низькорівневі програмні тести, що перевіряють окремі частини коду, наприклад, методи чи класи. Найчастіше вони зберігаються у тому ж репозиторії, що і сам програмний продукт, збираються та проганяються як частина збірки — це дозволяє відразу побачити чи пройшли тести.

Найбільша кількість тестів генерується саме для модульного тестування, а разом з інтеграційним вони займають більшу частину тест-кейсів на проекті.

Прогін тестів займає певний час. Іноді, коли є терміновий функціонал, і збірка має пройти якнайскоріше, критичним є виконання тестів у найкоротший проміжок часу.

Існують бібліотеки, які використовуються для модульного тестування і допомагають розробникам. Так, для мови Java найпопулярнішим інструментом є JUnit.

Основним його призначенням є прискорення та спрощення написання модульних тестів на Java. JUnit — це тестова система (framework), яка використовує анотації для виявлення методів, що визначають тест.

Новизна та аналіз існуючих рішень.Тести можуть виконуватися у різний спосіб. Їх можна запускати як послідовно — один за одним — так і паралельно — для того щоб вони виконувались незалежно від результату виконання інших тестів. На відміну від JUnit 4, який має parallelSuite, що дозволяє запускати тести паралельно, — у JUnit 5 такої можливості за замовчуванням немає. Саме тому пропонується рішення, яке зможе прискорити виконання тестів та використати ресурси раціонально.

Виділяють 2 підходи до розробки через тестування — Test Driven Development і Behavior Driven Development. Концепції обох підходів схожі — спочатку пишуть тести, а потім йде розробка програмного продукту. Проте TDD більше відноситься до програмування та тестування на рівні технічної реалізації продукта, коли розробники створюють тести. А от в BDD тести описуються звичайною мовою, описуючи поведінку(behaviour). Такий підхід допомагає зменшити кількість документації, особливо, застарілої.

Розробка через тестування зазвичай включає такі етапи[69]: написання тесту та перевірка (до тих пір, поки перевірка не покаже позитивний результат — тест пройдений), написання коду, запуск всіх тестів та перевірка (якщо тести не проходять, розробник повертається до етапу написання коду), після того, як всі тести пройдуть успішно — можна приступати до покращення коду (рефакторингу) з метою полегшити розуміння коду та майбутні зміни, видалити код, що дублюється. Далі цикл повторюється для наступних нових функціональностей.

69 Beck, K. Test-Driven Development by Example, Addison Wesley, 2003

У розробці через тестування активно використовуються mock-об'єкти — тип об'єктів, що реалізують задані аспекти модельованого програмного оточення.

Mock-об'єкт являє собою специфічну фіктивну реалізацію інтерфейсу, призначену виключно для тестування взаємодії, і щодо якої висловлюється твердження.

Тобто mocking — це спосіб перевірки функціональності класу в умовах відокремленості. Він не вимагає з'єднання з базою даних або властивостей читання файлів чи файлового сервера для перевірки функціональності. Mock-об'єкти виконують емуляцію справжньої служби.

Mockito — це платформа для тестування з відкритим кодом для Java, випущена в рамках ліцензії MIT. Структура дозволяє створювати тестові об'єкти для розробки на основі тестів (TDD) або керування поведінкою (BDD).

Фреймворк Mockito надає ряд можливостей для створення згаданих вище "моків" замість реальних класів або інтерфейсів при написанні JUnit тестів.

Найбільшого поширення набули такі можливості Mockito:

— створення mock-об'єктів для класів та інтерфейсів;

— перевірка виклика методу і значень параметрів переданих методу;

— використання концепції "часткової заглушки", при якій мок створюється на клас з визначенням поведінки, необхідної для деяких методів класу;

— підключення до реального класу "шпигуна" (spy) для контролю виклику методів.

Крім Mockito, використовують також EasyMock, що теж полегшує створення мок-об'єктів. Він використовує Java Reflection для створення "макету" об'єктів для певного інтерфейсу, а це — не що інше, як проксі для реальних реалізацій. Переваги EasyMock:

— немає введення вручну — не потрібно писати моки об'єктів самостійно;

— безпечний рефакторинг — перейменування імен методу інтерфейсу або параметри реорганізації не порушать тестовий код, оскільки мок-об'єкти створюються під час виконання;

— підтримка повернення значень;

— підтримка перевірки перевірки порядку викликів методу;

— підтримка анотацій[70].

Під час написання тестів іноді важко отримати рівновагу між перевизначенням тесту (і тим, що робить його чутливим до змін), та недостатньо точним визначенням (тоді тест стає менш цінним, оскільки він продовжує проходити навіть тоді, коли тестований об'єкт не працює). Маючи інструмент, який дозволяє точно вибирати тестовий аспект і описувати значення, які воно повинно мати, до контрольованого рівня точності, дуже допомагає в написанні тестів, які є "справедливими".

Для використання переваг та функцій модульного тестування в JUnit 5 впроваджені відповідні засоби реалізації. Так, в 4-ій версії використовуються engine, аннотації (annotations), рефлексія (reflection).

---

70 EasyMock Basics [Електронний ресурс]. — Режим доступу: https://www.tutorialspoint.com/easymock/easymock_overview.htm

ДОДАТОК Б

Програмні засоби прискорення модульного тестування

Акт впровадження

УКР.НТУУ “КПІ”.ТВ-з7127мп

Аркушів 2

2018